Zur Homepage der Dissertation

DEVELOPING MOBILE AGENTS THROUGH A FORMAL APPROACH

by

Andreea Barbu, Dipl.Inf.

Thesis prepared in french-german co-tutelle

Thèse	**Dissertation**
préparée en co-tutelle franco-allemande	in französisch-deutscher Doppelbetreuung erstellt
à	und dem
l'Université Paris 12	Department für Informatik
Val-de-Marne U.F.R. de Science	Carl v. Ossietzky Universität, Oldenburg
présentée pour obtenir	vorgelegt zur Erlangung
le grade de Docteur ès Sciences	des akademischen Grades eines Doktors
de l'Université Paris 12	der Naturwissenschaften

Verteidigt am 12 September 2005 vor der Prüfungskommission:

Eike Best	Direktor (G), Gutachter
Emmanuel Chailloux	Gutachter
Fabrice Mourlin	Gutachter
Ernst Rüdiger Olderog	Gutachter
Elisabeth Pelz	Direktorin (F)
Laure Petrucci	Gutachter

CO-TUTELLE

This thesis is written within the scope of a "co-tutelle" contract between the universities Paris12 Val de Marne (France), and C. v. Ossietzky, Oldenburg (Germany). The contract signed between the two universities requires that certain parts of the thesis (short abstracts and summaries) are given as well in French as in German. The preface gives a "Resumé" and a "Zusammenfassung" which introduce the subject of this thesis and give an overview of the contents.

Le contrat de thèse en co-tutelle signé entre l'Université Paris12 Val de Marne (France) et l'Université C. v. Ossietzky, Oldenburg (Allemagne) requiert certaines parties (résumés courts et résumés) en français et en allemand. Le résumé nous amène dans le sujet de cette thèse et donne une vue ensemble du contenu.

Der Vertrag über die Doppelbetreuung und gemeinsame Durchführung einer Promotion zwischen der Universität Paris12 Val de Marne (Frankreich) und C. v. Ossietzky, Oldenburg (Deutschland) sieht vor, daß bestimmte Teile der Arbeit (Kurzzusammenfassungen und Zusammenfassungen) in französicher und deutscher Sprache gegeben werden. Die Zusammenfassung führt in das Thema der Arbeit ein und gibt einen Überblick über ihren Inhalt.

SHORT ABSTRACT

This thesis deals with the modeling and validation of mobile agent systems. The development of a support structure for mobile agents demands the development of solutions for a set of specific problems that appear due to mobility.

A basic question in software development is if the proposed program is really a solution for the considered problem. One way to answer this question is through the use of formal methods. In our approach, the first step is to build a model of the solution (specification) using a formal language, the higher-order π-calculus. Having this formal model as a base, we can: validate the model through simulations; carry out mathematical tests to guarantee that this model possesses the required properties (verification); follow a rigorous software development, being able to prove that the implementation is correct with respect to the specification (generation of correct code).

In a second step we propose three different methods for a mobile agent system implementation. Making use of our results, we have implemented a prototype called *HOPiTool* which allows the possibility of validation of mobile agent systems conceived with higher-order π-calculus.

Keywords: Mobile agents, formal languages, π-calculus, modeling, validation, verification, concurrency, semantics.

RÉSUMÉ COURT

Nous nous intéressons dans cette thèse à la modélisation et à la vérification de systèmes d'agents mobiles. Le développement d'une structure de soutien pour les agents mobiles demande le développement de solutions pour un ensemble de problèmes spécifiques qui apparaissent en raison de la mobilité.

Une question fondamentale dans le développement de logiciel est: le programme proposé est-il vraiment une solution pour le problème considéré. Une façon de répondre à cette question consiste à utiliser les méthodes formelles. Dans notre approche, dans une première étape, nous construisons un modèle du problème (la spécification) en utilisant un langage formel, le π-calcul d'ordre supérieur. En ayant ce modèle formel comme base, nous pouvons réaliser des preuves mathématiques pour garantir que ce modèle possède les propriétés voulues (vérification); valider le modèle à travers des simulations; suivre le développement de logiciel rigoureux, étant capable de prouver que l'implémentation est cohérente par rapport à la spécification (génération de code correct).

Dans une deuxième étape nous proposons trois méthodes différentes pour l'implémentation d'un système d'agents mobiles. En profitant de nos résultats, nous avons implémenté un prototype (appelé *HOPiTool*) qui permet la validation de systèmes d'agents mobiles conçus avec le π-calcul d'ordre supérieur.

Mots clés: Agents mobiles, langage formel, π-calcul, modélisation, validation, vérification, concurrence, sémantique

KURZZUSAMMENFASSUNG

Diese Arbeit beschäftigt sich mit der Modellierung und Validierung von mobilen Agenten. Die Entwicklung einer Unterstützungsstruktur für mobile Agenten fordert die Entwicklung von Lösungen für eine Reihe spezifischer Probleme, die aufgrund der Mobilität erscheinen.

Eine grundlegende Frage in der Software-Entwicklung ist, ob das vorgeschlagene Programm wirklich eine Lösung für das Problem darstellt. Diese Frage können wir beantworten, indem wir von formellen Methoden Gebrauch machen. Der erste Schritt ist, ein Model des Problems (eine Spezifizierung) zu bauen, das eine formelle Sprache, den higher-order π-calculus einsetzt. Mit dem formellen Model als Basis, können wir mathematische Tests ausführen, um zu versichern, daß dieses Modell die erforderliche Eigenschaften besitzt (Verifizierung). Wir können durch Simulationen das Model validieren, und wir können zeigen, daß die Implementierung des Systems in Bezug auf die Spezifizierung korrekt ist (Generierung des richtigen Codes).

In einem zweiten Schritt schlagen wir drei verschiedene Methoden für die Realisierung eines mobilen Agenten-System vor. Von unseren Ergebnissen Gebrauch machend, haben wir einen Prototyp (genannt *HOPiTool*) implementiert, der die Möglichkeit der Validierung von mobilen Agenten Systeme die mit dem higher-order π-calculus spezifiziert, anbietet.

Schlüsselwörter: Mobile Agenten, formelle Sprachen, π-Kalkül, Modellierung, Validierung und Verifikation, Nebenläufigkeit, Semantik.

ACKNOWLEDGEMENTS

"To make an end is to make a beginning.
The end is where we start from."
T.S. Eliot

I would like to express my deepest gratitude to all of those who gave me the possibility to complete this thesis. First of all, I would like to thank Prof. Elisabeth Pelz, my "Doktormutter", for having been mentor and confidant for me every single step of the way, from the tedious task of relocating me from Oldenburg to Paris, through valiantly fighting my administrative battles, to coaching me in the art of scientific and methodical research and project development.

Many thanks go to my "Doktorvater", Prof. Eike Best, for giving me the wonderful and life changing opportunity to do this co-tutelle, in the first place, for appraising my work, managing and supporting my research from across the border on the German side, and for keeping an eye on my progress and being there with advice and/or help when I needed it.

I would like to express my deepest and sincerest gratitude to my principal "encadrant", Dr. Fabrice Mourlin. His wealth of knowledge and his logical thinking have been invaluable for me, as have his understanding, encouragement and personal guidance provided a sound basis for the present thesis. Thanks to him I have learned so much and he always lent me the strength to carry on with his energy and ideas.

To Dr. Hans Fleischhack, I would like to extend my appreciation and gratitude for having gotten me started on the path that led to my research and this thesis.

I would like to thank my thesis reviewers, Prof. Laure Petrucci and Dr. Emmanuel Chailloux, for accepting to read my work, for their helpful comments and for being part of the jury committee. I owe gratitude to Prof. Ernst-Rdiger Olderog for also accepting to participate in the jury.

I am very grateful to my husband for his emotional support, an ever open ear and mind and for spending so many weekends on revising the English of my manuscript.

I am also grateful to my colleague Stefan for helping me with his knowledge in program development. His comments, propositions and questions have been greatly helpful for me.

My warmest thanks go to Katharina, because: "A problem shared is a problem halved", for her moral support, for her remarkable conclusions and for her friendship which helped me get through this time.

I am also very thankful to my friend, Christine, for helping me to translate the first chapter into correct (and legible) French.

This thesis is dedicated to my parents and my brother, but especially to my "mam", who always believed in me and gave me the strongest possible of support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Preface

Introduction

De nos jours, le développement rapide de notre société d'information a généré une explosion du réseau informatique. En témoigne en particulier la constante augmentation du nombre d'utilisateurs de réseaux, en particulier d'Internet. Les technologies dans le secteur de l'électronique grand public offrent soit un accès rapide et simple à Internet, soit la possibilité d'une connexion "à la volée" via les réseaux GSM (Global System for Mobile Communication). En outre, la quantité de ce type de services croît à un rythme effréné.
En lien avec cette évolution dans le domaine des réseaux, une tendance suscite plus particulièrement de nouvelles interrogations: l'informatique mobile. Non seulement les ordinateurs portables sont intégrés aux réseaux informatiques, mais également les téléphones mobiles qui nous ouvrent l'accès aux réseaux via "Wireless Application Protocol" (WAP) et "Wireless Markup Language" (WML). Les "Personal Data Assistants" (PDA) sont une option supplémentaire pour établir des connexions aux réseaux mobiles. Ils utilisent des techniques de connexion comme "Infrared Data Association" (IrDA) , Bluetooth ou encore, comme c'est le cas pour les téléphones mobiles, le GSM et "Universal Mobile Telecommunications System" (UMTS). Les applications mentionnées ci-dessus sont opérationnelles sur des réseaux non restreints dynamiquement, ce qui signifie que des agents mobiles peuvent être ajoutés à tout moment de l'opération. Les services utilisés pour ces applications peuvent aussi utiliser d'autres services,

- dont la durée de vie est limitée dans le temps ou octroyant des droits d'accès à durée variable.
- dont les fournisseurs se perfectionnent. Par exemple, le service d'un annuaire téléphonique peut être sur une machine "A" et pour des problèmes divers dus à "A", être déplacé sur une machine "B" plus perfectionnée.

La propriété centrale de ces applications est donc que les réseaux sont progressifs.
Ce développement a donné lieu à de nombreux problèmes et de nouveaux défis, défis dont nous n'analyserons ici qu'un certain nombre: la croissance de la capacité du réseau n'a pas pu jusqu'à présent pallier au nombre en rapide augmentation d'utilisateurs. Bien que l'on ait vanté Internet pour son réseau aux possibilités inépuisables, il ne peut offrir qu'un catalogue restreint de services et d'usages, sans qu'un ordinateur y soit connecté en permanence. La récente émergence de la technologie des agents mobiles apparaît comme un moyen de résoudre ce problème.
Les agents mobiles sont des programmes qui peuvent se déplacer (ou "migrer") d'un ordinateur à l'autre au sein d'un réseau et exécuter de manière autonome les commandes de leurs possesseurs. Cette "migration" rend possible l'utilisation de l'ensemble des ressources de l'ordinateur membre au sein d'un réseau local et l'exécution efficiente d'une tâche donnée au moyen du traitement coordonné du problème par les agents membres. Par exemple, un agent "A" récupère un fichier de données C.txt sur une machine "C", puis un fichier de données D.txt sur une machine "D". La "localité" caractéristique est ici essentielle pour améliorer la fiabilité des applications. Il est plus avantageux qu'un agent se déplace afin de collecter des fichiers de données plutôt qu'il en fasse une lecture à distance. Les problèmes inhérents au réseau se trouvent réduits et la sécurité accrue par cette utilisation d'agents mobiles.
Si l'on tentait de définir ces agents mobiles au moyen d'un exemple de la vie quotidienne, on pourrait imaginer le scénario suivant:
Supposons que Bob veuille se rendre au cinéma, sans toutefois savoir où son film favori sera projeté ce soir. Par chance, Bob a un petit frère Jake qui se fait un plaisir de jouer à l'agent 007 pour lui. Jake a donc pour tâche de trouver où le film favori de Bob va être projeté. Pour mener à bien sa mission, Jake doit se rendre dans chaque cinéma du voisinage et, si possible, en revenir avec le ticket espéré. Après s'être acquitté de cette tâche, il peut choisir de rentrer à la maison faire son compte-rendu ou aller jouer avec ses amis "agents secrets". Le jeune 007 se sent investi d'une mission importante et passe à l'action. Il décide cependant de s'épargner les allées et venues grâce à l'aide de ses amis agents. Il les appelle et leur confie la tâche de se rendre au cinéma le plus proche et de s'y renseigner sur le film préféré de Bob. Après que chacun se soit acquitté de son compte-rendu, Jake assemble les résultats et transmet l'information à Bob.
Transposé au domaine de la technologie, de l'Internet et particulièrement du World Wide Web (WWW), cet exemple prendrait l'aspect suivant:
Un agent mobile a pour tâche de rassembler à partir d'une sélection de critères des données de n'importe quelles source de données (par exemple des serveurs WWW). L'agent migrerait alors vers les ordinateurs, effectuerait des recherches dans leurs sources de données et transmettrait les résultats à l'utilisateur, au lieu de simplement charger et

examiner les pages des serveurs à travers internet. Les résultats de recherche de la thèse de Theilmann [96] montrent que ce procédé peut permettre un gain de temps, aussi bien au niveau de la durée de connexion au réseau que de l'accomplissement de la tâche, selon l'effort fourni par l'agent et l'étendue des données à explorer et retransmettre au possesseur. Ces avantages, doublés de la capacité des agents de s'acquitter de leur tâche de manière asynchrone sans aucun contact supplémentaire avec leurs possesseurs, les rendent particulièrement intéressants à utiliser avec des outils terminaux.
La recherche et le filtrage dans des sources de données ne sont pas les seules applications que les agents mobiles sont à même de mener à bien. D'autres applications existent dans les domaines du commerce électronique, de l'administration des réseaux, de la distribution de l'information d'ordinateurs parallèles, de services comme WebLogic qui recourt à des agents mobiles pour détecter de nouveaux services sur le serveur, et de la détection des intrusions. Diverses publications ont évoqué bien d'autres champs d'application, comme [72], [46], [94]. Les cinq à six dernières années ont été témoin de quantité de tentatives de mise en pratique de cette technologie, quand bien même celles-ci apparaissent à présent peu abouties au regard du niveau de la recherche menée actuellement. Le projet "Agents for Remote Action" (Ara) (1997) [84] est une plate-forme pour une exécution portable et sûre d'agents mobiles, développée par l'Université de Kaiserslautern. Le but spécifique de Ara, à la différence de plate-formes similaires, consiste à garantir la fonctionnalité totale des agents mobiles, tout en conservant autant que possible les modèles de programmation et les langages établis. Les prototypes entre temps issus de la recherche sont, pour n'en citer que quelques-uns, D'Agents [49] (dernière version en 2002), MadKit [51] de l'Université de Montpellier (France) et Mole [9] (achevé avec succès au bout de 5 années en 2000). Le projet Mole à Stuttgart, consacré aux agents mobiles (MA), les réexaminait dans le contexte d'alors et testait en particulier leur fiabilité, sécurité et le mélange de Java et CORBA au sein du même système. D'autres produits ont trouvé des débouchés commerciaux, tels Aglets [61] (IBM fournit déjà de véritables services obtenus à partir de ce projet, comme TabiCan sur Internet), Grashopper [10] (cette plate-forme de développement d'agent initiée par IKV++ en aot 1998, permet à l'utilisateur de créer une profusion d'applications fondées sur la technologie des agents), LEAP [6] (ce projet s'attaque au besoin en infrastructures ouvertes et services qui soutiennent les entreprises dynamiques et mobiles) ou Java Agent DEvelopment Framework (JADE) [11], une plate-forme logicielle pour développer des applications basées sur les agents en accord avec les spécifications pour les systèmes interopérationnels intelligents à agents multiples de FIPA [39]). Tous ces exemples sont des plate-formes très techniques, dont le développeur a besoin pour développer ses agents mobiles.
Bien que beaucoup d'obstacles technologiques aient été surmontés grâce à la vitesse

plus rapide de transfert en réseau, l'utilisation de machines virtuelles et de langages de programmation indépendants de la plate-forme (à savoir Java, HTML, XML, PostScript, etc....) et d'ordinateurs plus rapides, efficaces et robustes, certains secteurs restent problématiques:

Les mécanismes de contrôle: Ce domaine est principalement aux prises avec le problème de la localisation et de l'arrêt des agents mobiles. Baumann donne un aperçu ([8]) d'une méthode mise en oeuvre et propose de nouvelles solutions pour ces tâches.

La sécurité: Ce domaine concerne la protection aussi bien de l'ordinateur que des agents mobiles contre les attaques d'agents mobiles et/ou d'environnements d'implémentation malveillants. F. Hohl a étudié ce problème dans [53].

La tolérance aux fautes: La tolérance aux fautes dans la programmation d'agents mobiles cherche à garantir, entre autre, que des agents mobiles ne soient pas bloqués ou perdus suite à leurs interactions avec d'autres agents mobiles ou ordinateurs, ou dans le cas d'une panne de réseau. Une solution à ce problème est examinée par M. Strasser dans [94].

ainsi que la large diffusion géographique, les dynamiques de réseau élargi afin, par exemple, de gérer des services d'un type nouveau. Un autre problème significatif réside dans la difficulté de tester des agents mobiles là où il est difficile de déterminer si des fautes ont été générées par les agents eux-mêmes ou par un autre élément du milieu, comme c'est en particulier le cas sur internet. A cela s'ajoute qu'il reste à définir le critère du test concernant la navigation (un agent est-il d'abord passé par un noeud C puis par un noeud D?) et l'activité locale d'un agent (pour lire le fichier $C.txt$ sur le noeud C, utilise-t-il les droits locaux ou un accès à distance?).

Motivation et contribution scientifique

Le travail présenté dans cette thèse se fixe pour but de proposer une technique pour l'élaboration et l'analyse de systèmes d'agents mobiles et de tenter de combler la brèche entre les modèles formels d'agents mobiles et les implémentations d'agents mobiles.
Si l'on considère l'exemple donné concernant la tâche d'investigation sur les heures de projection de film confiée au petit frère de Bob, quelques questions pourraient se poser durant cette mission: Jake, le petit frère, a-t-il compris que le ticket était requis pour le soir même? Avait-il l'argent pour l'acheter? A-t-il fait part à ses amis du souhait de son frère d'acheter ce ticket? Que se serait-il passé si, après l'achat du ticket, Jake avait

décidé de ne pas rentrer directement à la maison, mais de jouer d'abord avec ses amis et avait alors perdu le ticket? Ces questions se posent aussi longtemps que la mission n'a pas été définie clairement. C'est seulement après avoir clairement établi quel est l'objectif final et l'enchaînement précis des événements que le jeune 007 peut être assuré de la bonne issue de sa mission. L'exemple décrit est une spécification naturelle et le point de départ à partir duquel nous devons réaliser un prototype et une simulation si nous voulons trouver un vrai cinéma. Pour ce faire, un grand nombre de solutions est envisageables. Etant donné que la spécification est textuelle, c'est-à-dire non formelle, il est impossible d'obtenir un prototype complet capable de réussir cette mission.
Ces questions doivent être prises en considération dans la conception des agents mobiles. La spécification pour les agents et leurs tâches a-t-elle été correctement effectuée? Le terme "spécification" est défini dans le Merriam-Webster-Online-Dictionary `http//:www.merriam-webster.com` comme "a detailed precise presentation of something or of a plan or proposal for something", c'est-à-dire comme "une présentation précise et détaillée de quelque chose ou d'un plan ou d'une proposition pour quelque chose". Nous nous concentrerons ici sur la "spécification formelle" car, présentée au sein d'une structure mathématique, elle permet d'établir des preuves ou des analyses et raisonnements formels.
L'axe majeur de notre réflexion porte sur les points suivants:

- Il donne un aperçu des projets internationaux sur les "agents mobiles" actuellement à l'essai et étudie les avantages et inconvénients des agents mobiles. (Chapitre 2)
- Il analyse différents langages de spécification formelle des agents mobiles. (Chapitre 3)
- Il présente et élargit une méthode où les spécifications formelles sont écrites dans le langage processus-algébrique "Higher-Order-Calculus" et vérifient quelques propriétés, en utilisant un model-checker existant. (Chapitre 4)
- Il propose trois différentes méthodes pour l'implémentation d'un agent mobile et étudie leur faisabilité pour choisir la méthode la plus adéquate. (Chapitre 5)
- Après la présentation d'une base théorique qui rend possible le développement d'un système d'agents mobiles, nous développerons un prototype *HOPiTool* (abréviation pour "Higher-Order-Calculus Tool"), permettant d'effectuer localement simulations et tests sur des agents mobiles avant leur mise en oeuvre au sein d'un réseau. (Chapitre 6)

Les défis de la standardisation

Les idées concernant le code mobile et son implémentation sont nées d'une approche pratique. La plupart des standards, plate-formes et langages actuellement disponibles et largement utilisés pour le développement de ce genre de systèmes témoigne de la réalité suivante: ils ont plutôt été élaborés ad-hoc que sur la base de recherches théoriques. Des modèles théoriques de calcul aussi bien que des langages de spécification formelle correspondants demeurent indispensables pour donner à ces idées un véritable fondement permettant une meilleure compréhension (ou clarification des concepts), une meilleure analyse, comparaison avec d'autres modèles et jusqu'à une prévision de leur évolution. Sur la base du π-calcul, des efforts ont été fournis dans le domaine des modèles informatiques des systèmes mobiles, en partant par exemple de machines à états abstraites [16] et d'environnements mobiles [23]. Dans tous les cas, le cadre de l'application pratique requiert aussi bien des langages de spécification de haut niveau que des langages de programmation dont la sémantique puisse être décrite en utilisant ces modèles. Il existe quelques projets de langages de programmation adaptés (par exemple KLAIM [75], Mobile UNITY [66], Pict [86], Nomadic Pict [104]) mais, au niveau de la spécification et de la validation/simulation d'applications mobiles, il n'existe toujours pas de méthode formelle largement utilisée pour construire des système pratiques. Au regard des systèmes de code mobile, les aspects de distribution complexes comme la localisation et la mobilité, la communication et, dans certains cas, les erreurs, ne sont pas de simples problèmes d'implémentation mais font partie des fonctionnalités du système. Sous les auspices des organes de standardisation comme MASIF [28], FIPA [41], CORBA [85], etc., des comités de standardisation travaillent constamment à l'établissement de standards pour les agents et en particulier pour les produits d'agents mobiles, pour lesquels la logique est d'être implémentée. Dans les premières phases de ces processus, beaucoup de caractéristiques, services et fonctionnalités, sont démontrés par des descriptions informelles opérationnelles, des tableaux et des visuels. Ces descriptions évoluent dynamiquement et leur traitement et validation deviennent dès lors rapidement difficiles.
Dans ce contexte, il faut se poser les questions suivantes:

- Pendant le design des systèmes et des services dans les stages initiaux, la discussion pourrait se concentrer sur un niveau de détail trop bas par rapport aux connaissances (données, messages, composants, etc ...) disponibles à ce moment là. Les descriptions qui incluent des détails sans intérêt ont tendances à obscurcir l'idée centrale qui est derrière une caractéristique, un service ou une fonctionnalité, surtout quand cette dernière à besoin d'être modifiée ou raffinée. Différents niveaux de détails (d'abstractions) sont souvent mélangés dans une seule descrip-

tion. Par exemple, l'énumération des nœuds qu'un agent doit explorer peut être déclarée mais aussi la manière de trouver le prochain nœud (si la route à suivre n'est pas connue au début de la mission).

- Il peut y avoir des ambigutés, des incohérences ou des interactions à l'intérieur ou entre les descriptions des services, ou entre les niveaux d'abstraction d'un service donné. Elles demeurent difficiles à repérer avec les méthodes conventionnelles d'inspection, et souvent restent cachées jusqu'à ce que les erreurs soient découvertes après l'implémentation, à un moment où les corrections peuvent être très coûteuses. Par exemple, dans le même cas la structure des messages entre les agents est plus importante que la structure des requêtes qui sont envoyées à l'hôte, quand l'agent veut savoir quels fichiers doivent être récupérés.

Une proposition pratique

Le processus d'aller de prérequis informels fonctionnels ou opérationnels à une spécification formelle de haut niveau a fait l'objet de nombreuses recherches. Cependant beaucoup de questions demeurent en suspens, en particulier en ce qui concerne les techniques les plus récentes de définition des prérequis et des spécifications formelles. Des techniques formelles de description ("Formal Description Techniques" (FDT)), comme le "Higher-Order π-Calculus" (HOπ) [90], ont été créés afin d'exprimer de manière formelle les prérequis fonctionnels. Ils sont particulièrement appropriés pour une définition précise des systèmes mobiles. Ces quelques dernières années, les milieux universitaire et industriel se sont tous deux penchés avec intérêt sur l'utilisation de scénarios pour élaborer des systèmes. Pourtant, un grand nombre de significations ont été attribuées au terme "scénario". Il a été associé à des traces (d'évènements internes ou externes), à la communication entre des composants, à des séquences d'interaction entre un système et son utilisateur, à des collections plus ou moins génériques de telles traces, etc. Des nombreuses notations sont aussi utilisées pour les décrire: des grammaires, des automates, des algèbres de processus et des diagrammes de communication.
Plusieurs techniques peuvent être utilisées pour s'attaquer aux problèmes liés aux processus de standardisation et aux scénarios. Nous utilisons l'algèbre de processus HOπ pour décrire la spécification des systèmes d'agents mobiles. Le langage de spécification (ou plutôt sa force d'expression), est directement lié aux contraintes des propriétés que l'on souhaite mettre au point. Par exemple, pour tenir compte des propriétés de performances, il est essentiel d'utiliser un langage de spécification qui supporte ces contraintes. Dans notre cas, on s'intéresse à la mobilité et on veut établir des propriétés montrant que les agents se déplacent correctement d'une machine "E" à une machine "F", il est

donc nécessaire que le langage de spécification permette de tels mouvements. Le "Higher Order π-Calculus" est un langage de spécification qui prend en compte l'aspect de la mobilité.
Après le travail préliminaire de sélection et d'amélioration de la structure la plus adéquate, nous développerons au niveau pratique un prototype *HOPiTool* pour la validation et le test à partir de scénarios, afin de détecter dès que possible les irrégularités, les ambiguïtés, les incomplétudes et autres problèmes. L'un des résultats pratiques requis par les développeurs est la mise au point d'une documentation aisément compréhensible et validée autant que possible, et la production d'une série validée de tests, qui puisse être réutilisée lors de phases ultérieures incluant l'implémentation. Nous illustrerons une partie de ce processus par un cas d'étude impliquant le protocole de localisation de service "Service Location Protocol" (SLP) [52]:

Proposition d'un scénario pour le "HOPiTool"

L'utilisation d'un langage formel de premier ordre comme le HOπ dans une approche fondée sur un scénario constitue un choix judicieux pour la description de systèmes mobiles et communicants. Elle est bien adaptée à la méthode de conception que nous proposons dans le schéma 1, où nous avons l'intention de combler la brèche entre les prérequis informels et la première conception d'un système. Les prérequis ont habituelle-

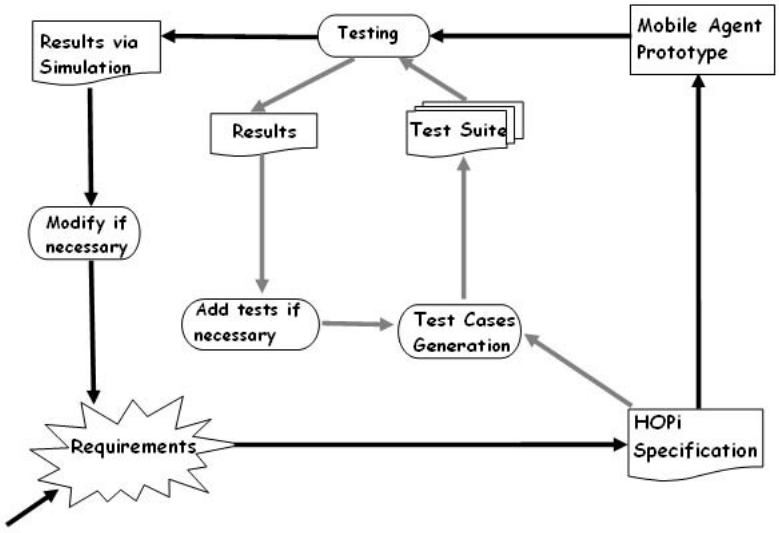


Figure 1: Approche fondée sur un scénario

ment un caractère évolutif et dynamique, ils se modifient et s'adaptent dans le temps.

De ce fait, un processus itératif et incrémental va permettre l'élaboration rapide de prototypes et la génération directe de tests à partir de scénarios. Le schéma 1.1 propose une méthode où le cycle principal est en charge de la description des scénarios. Ceux-ci sont ensuite fusionnés afin de synthétiser (manuellement) une spécification de premier ordre qui sera le point de départ pour la construction de notre prototype. Dans un même temps, des cas tests peuvent être directement générés par ces scénarios, puis être utilisées pour tester le prototype par rapport à sa spécification. Les résultats que nous pouvons obtenir de ces tests, peuvent nous permettre de savoir si des cas tests supplémentaires sont nécessaires pour finir de couvrir la spécification souhaitée. Afin de vérifier et d'analyser les propriétés de notre système, il est nécessaire d'insérer un "model checker" disponible (comme UPPAAL) au sein de notre prototype. Nous partons ensuite de l'hypothèse que le prototype correspond aux prérequis. Nous avons observé plusieurs avantages à cette méthode:

- La séparation des fonctionnalités de la structure sous-jacente: étant donné que les scénarios sont formalisés à un niveau d'abstraction supérieur aux échanges de messages, diverses structures ou architectures sous-jacentes peuvent être évaluées avec plus de flexibilité. Les scénarios deviennent dès lors des entités fortement réutilisables. Comme nous le mentionnerons plus loin, ils peuvent être réutilisés pour tester l'implémentation.

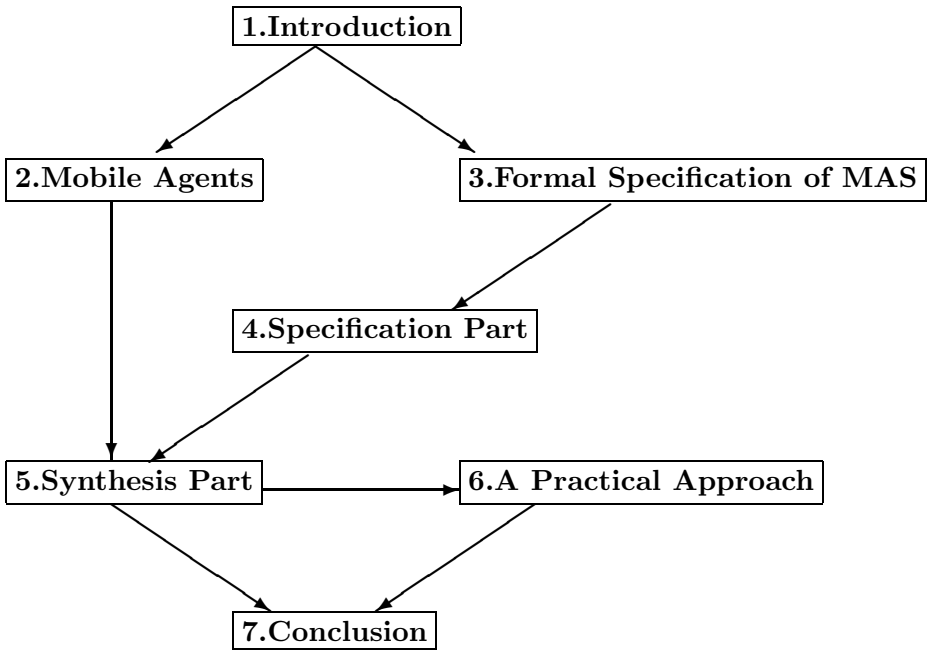
- le prototypage rapide: dès que la structure et le scénario sont sélectionnés et analysés, un prototype peut être rapidement produit.

- La documentation de la conception: la documentation est produite tout au long de la phase de conception. Très souvent, les concepteurs ne produisent une documentation sur leur conception que s'ils y sont obligés; cet angle d'approche encourage le concepteur à produire méthodiquement une documentation utilisable.

Les applications des agents mobiles, écrites conformément au langage formel HOπ et à la méthode indiquée précédemment, peuvent être simulées au moyen du prototype *HOPiTool*, de la même manière qu'un code peut être généré automatiquement pour eux en faisant un mapping direct en classes Java, en prenant en compte l'usage d'une plate-forme support de mobilité, comme "Java Intelligent Network Infrastructure" (Jini).
Le *HOPiTool* est un prototype pour la simulation des agents mobiles et peut être caractérisé par le mapping d'une spécification "higher-order π-Calculus" en un modèle de simulation et la génération d'un code pour une plate-forme support de mobilité.

Description du contenu

Le chapitre 2 présente différents systèmes d'agents mobiles ainsi que plusieurs projets internationaux les concernant. Ce travail traite uniquement des projets actuels à partir de l'année 2000. Le choix de ces projets montre la diversité des applications qui recourent à des agents mobiles. Le sujet des agents mobiles est introduit par l'analyse de diverses interprétations du terme "agents mobiles", esquissant les possibles applications des agents mobiles et présentant leur définition telle qu'elle sera abordée dans ce travail. Le chapitre 3 va ensuite se concentrer sur les différentes méthodes qui seront utilisées dans la spécification des agents mobiles. Le chapitre 4 analyse une méthode formelle de premier ordre pour la mobilité et l'utilisation d'un model-checker qui vérifie certaines propriétés de système d'un protocole de mobilité. Le chapitre 5 propose trois différentes méthodes pour implémenter au mieux un système d'agents mobiles et analyse la faisabilité d'une application pratique. Le chapitre 6 propose une application pratique étayée par notre travail théorique préalable. Le prototype *HOPiTool* sera introduit et une étude de cas sera analysée. Le chapitre 7 clôt le travail par un récapitulatif et une analyse des développements potentiels à venir. Le schéma ci-après met en évidence les liens entre les différents chapitres.



Einführung

Die rasante Entwicklung unserer Informationsgesellschaft hat inzwischen zu einer explosionsartigen Evolution der Computervernetzung geführt. Dies zeigt sich besonders im Internet an einer ständig wachsenden Benutzerzahl. Neue Technologien im Anwendungsbereich bieten längst sowohl einen schnellen und unkomplizierten Zugang ins Internet als auch die Möglichkeit der "on-the-go"-Verbindung via GSM in die Netzwerke der mobilen Kommunikation. Das vielfältige Angebot an derartigen Dienstleistungen wächst in immer kürzeren Abständen.
Ein besonderer Trend, der im Zusammenhang mit dieser Evolution im Netzwerkbereich steht und neue Fragen aufwirft, ist die Mobile Datenverarbeitung. Nicht nur Laptops werden in Rechnernetze integriert, auch mobile Telefone eröffnen uns über "Wireless Application Protocol" (WAP) und "Wireless Markup Language" (WML) den Zugang. Personal organizers oder PDA's ergänzen das Spektrum der Möglichkeiten, Verbindungen in mobile Netze herzustellen. Für den Aufbau der Verbindung werden Techniken wie "Infrared Data Association" (IrDA), "Bluetooth" oder im Falle der Mobiltelefone auch "GSM" und "Universal Mobile Telecommunications System" (UMTS) genutzt. Diese Anwendungen operieren auf der Basis nicht dynamisch beschränkter Netze, d.h. mobile Agenten können während der Ausführung jederzeit hinzugefügt werden.
Die immer höhere Netzwerk-Kapazität genügt nach wie vor nicht den Ansprüchen einer weiter wachsenden Zahl von Benutzern. Obwohl das Internet für seine unerschöpflichen Möglichkeiten angepriesen wird, kann es nur einen beschränkten Katalog von Dienstleistungen bedienen. Einen Weg, dieses Problem zu lösen, stellt die Anwendung der jüngst sich entwickelnden Technologie mobiler Agenten in Aussicht.
Mobile Agenten sind Programme, die sich innerhalb eines Netzes von einem Computer zu einem anderen bewegen (oder wie man sagt: "migrieren") und Befehle im Auftrag ihrer Eigentümer autonom ausführen können. Diese "Migration" ermöglicht die gegenseitige Benutzung sämtlicher Computerressourcen innerhalb des lokalen Netzes und die effiziente Ausführung einer festgelegten Aufgabe mittels einer kooperativen Problem-Verarbeitung, die Mitagenten koordiniert.
Ein Agent "A" sammelt beispielsweise eine Datei C.txt auf einer Maschine "C", dann eine Datei D.txt auf einer Maschine "D". Das "Orts-Merkmal" ist dabei wesentlich, denn es bewirkt eine zuverlässigere Anwendung. Es ist vorteilhafter, wenn ein Agent sich an den Ort der Dateien bewegt, um sie zu sammeln, als wenn er diese aus der Entfernung liest. Die Anwendung von mobilen Agenten vermindert Netzwerkprobleme bei gleichzeitiger Steigerung der Sicherheit.
Um die Definition der mobilen Agenten mit einem Beispiel aus dem täglichen Leben anschaulich zu machen, stelle man sich das folgende Szenario vor:

Angenommen, Bob möchte ins Kino gehen, ohne jedoch zu wissen, wo sein Lieblingsfilm am Abend gezeigt wird. Glücklicherweise hat Bob seinen kleinen Bruder Jake, der gerne den Agenten 007 für ihn spielt. Jake bekommt also die Aufgabe herauszufinden, wo Bobs Lieblingsfilm gezeigt wird. Um seine Mission zu erfüllen, soll Jake zu jedem Kino in der Nachbarschaft gehen und wenn möglich eine Karte für den Lieblingsfilm besorgen. Nach der Erfüllung dieser Arbeit kann er dann je nach Belieben entweder zu Hause über seine Ergebnisse berichten oder mit seinen "Geheimagent"-Freunden spielen gehen. Jake hält die Mission für seiner würdig und setzt sich in Aktion. Er entscheidet jedoch, sich die Lauferei zu sparen und die Hilfe seiner Agentenfreunde anzunehmen. Er ruft sie an und bittet jeden, zum jeweils nächsten Kino zu gehen und sich nach Bob's Lieblingsfilm zu erkundigen. Wenn alle ihre Berichte abgeliefert haben, wird Jake die Ergebnisse zusammenstellen und die Information an Bob weiterleiten.
Auf den Bereich der Technologie, des Internets und vor allem des WWW. übertragen, würde das Beispiel so klingen:
Ein mobiler Agent bekommt die Aufgabe, anhand einer Auswahl an Kriterien definierte Daten von verfügbaren Datenquellen (z.B WWW-Server) zu sammeln. Der Agent würde dann zu den Computern migrieren, ihre Datenquellen erforschen und die Ergebnisse an den Benutzer übermitteln, anstatt lediglich aus dem Internet die Seiten mit dem fraglichen Inhalt von den Servern zu laden und zu durchsuchen. Die Forschungsergebnisse von Theilmann's Arbeit [96] zeigen, daß dieses Verfahren Zeit sowohl hinsichtlich der Aufenthaltsdauer im Netz als auch bei der Erledigung der Aufgabe spart, je nach Anstrengung (effort) seitens des Agenten und dem Umfang der Daten, die zum Besitzer zurückübertragen werden sollen. Über diese Vorteile hinaus macht besonders die Fähigkeit zur asynchronen und vom Kontakt mit ihrem Eigentümer unabhängigen Auftragserledigung Agenten besonders interessant für Anwendungen mit end devices.
Nicht nur das Durchsuchen und Filtern von Datenquellen kann von Mobilen Agenten übernommen werden. Andere Anwendungen finden sich in den Bereichen des E-commerce, der Netzwerkverwaltung, bei der Informationsverteilung von parallelen Computern, bei Dienstleistungen wie dem WebLogic, die auf mobile Agenten zurückgreifen, um neue Dienste auf dem Server oder Fremdeindringen ausfindig zu machen. Verschiedene Publikationen erwähnen Anwendungsgebiete über diese genannten hinaus, so [72] oder [46] oder [94]. In den letzten fünf, sechs Jahren hat es mehr und mehr Versuche einer Umsetzung der Technologie in die Praxis gegeben, wenn sie auch im Vergleich zum Niveau der bis heute geleisteten Forschung eher gering ausfallen. Das "Agents for Remote Action"-Projekt (Ara) (1997) [84] ist eine Plattform für eine tragbare und sichere Erprobung von mobilen Agenten, die an der Universität von Kaiserslautern entwickelt wird. Das spezifische Ziel von Ara im Unterschied zu ähnlichen Plattformen besteht darin, die umfassende Funktionalität mobiler Agenten unter weitestgehender

Beibehaltung der feststehenden Programmiermodelle und Sprachen zu garantieren. Aus der Forschung inzwischen hervorgegangene Prototypen sind, um nur einige zu nennen, D'Agents [49] (letzte Version in 2002), MadKit [51] von der Universität von Montpellier (Frankreich) und Mole [9] (erfolgreich beendet nach 5 Jahren in 2000). Auch das Mole-Projekt widmete sich Mobilen Agenten (MA) und prüfte diese insbesondere auf Zuverlässigkeit, Sicherheit und hinsichtlich einer Anwendung von Java und CORBA innerhalb desselben Systems. Zu weiteren, kommerzialisierten, Produkte gehören Aglets [61] (IBM bietet bereits Dienstleistungen an, wie TabiCan im Internet), Grashopper [10] (diese durch IKV im August 1998 gestartete Agent-Entwicklungsplattform ermöglicht dem Benutzer eine Vielzahl von Anwendungen auf der Basis der Agenten-Technologie), LEAP [6] (dieses Projekt konzentriert sich auf den Bedarf an offenen Infrastrukturen und Dienstleistungen, welche dynamische, mobile Unternehmen unterstützen) oder JADE [11] (Agent DEvelopment Framework, eine Software-Plattform zur Entwicklung Agentenbasierter Anwendungen in Übereinstimmung mit den Spezifizierungen für intelligente Multiagenten-Systeme der FIPA [39]).
Obwohl viele technologische Hürden, wie schnellere Netz-Transferraten, der Einsatz von virtuellen Maschinen und plattformunabhängigen Programmiersprachen (d. h. Java, HTML, XML, PostScript, etc. ...) sowie schnellere, bessere und robustere Computer, inzwischen genommen sind, bleiben weiterhin bestimmte Problemfelder bestehen wie:

Kontrollmechanismen: Dieses Gebiet ist hauptsächlich mit dem Problem der Lokalisierung und der Terminierung mobiler Agenten beschäftigt. Baumann gibt eine Übersicht ([8]) der existierenden Methoden und stellt weitere Lösungen für diese Aufgaben in Aussicht.

Sicherheit: Dieses Gebiet befaßt sich mit dem Schutz sowohl des Computers als auch der mobilen Agenten gegen Angriffe von böswilligen mobilen Agenten und/oder Programmierumgebungen. F. Hohl in [53] hat dieses Problem untersucht.

Fehler-Toleranz: Die in die mobilen Agenten implementierte Fehler-Toleranz soll u.a. sicherstellen, daß mobile Agenten nicht blockiert oder aufgrund ihrer Wechselwirkung mit anderen mobilen Agenten oder Computern oder im Falle eines Netz-Stillstands nicht verloren gehen. Eine Lösung für dieses Problem wird in [94] durch die M. Strasser besprochen.

sowie die weite geographische Verbreitung, die vergrößerte Netz-Dynamik, um beispielsweise mit Dienstleistungen neuer Art umzugehen. Ein weiteres erhebliches Problem besteht in der Schwierigkeit, mobile Agenten dort zu testen, wo, wie besonders innerhalb des Internets, es unklar ist, ob Fehler durch die Agenten selbst oder die Umgebung

verursacht werden. Hinzu kommt, daß die Testkriterien durch die Navigation (also: ist ein Agenten zuerst durch einen Knoten "C" und dann durch "D" gegangen?) und die lokale Tätigkeit eines Agenten (also: nutzt dieser, um die Datei "C.txt" an dem Knoten "C" zu lesen, die lokalen Rechte oder einen Fernzugriff (remote access)?) definiert werden müssen.

Motivation und wissenschaftlicher Beitrag

Die hier vorgestellte Forschungsarbeit zielt auf eine Technik für den Entwurf und die Analyse mobiler Agenten-Systeme. Die Arbeit versucht, die Lücke zwischen formellen Modellen mobiler Agenten und ihrer Anwendung/Implementierung zu überbrücken.
Auf das oben gegebene Beispiel mit den Protagonisten Jake und Bob und der Aufgabe, Kinoprogramme zu erfragen, zurückkommend, sollen einige Fragen nachgetragen werden, die sich während Jakes Mission ergeben könnten: Hat Jake, der kleine Bruder, verstanden, daß die Karte für den selben Abend gewünscht wurde? Hatte er Geld, um eine Karte zu kaufen? Würde er seinen Freunden den Wunsch des Bruders, die Kinokarte zu kaufen, mitteilen? Was hätte geschehen sollen, wenn ihr Geld nicht ausgereicht hätte? Was wäre geschehen, wenn Jake nach dem Kauf der Karte entschieden hätte, nicht gleich nach Hause zu gehen, sondern erst mit seinen Freunden zu spielen und dabei die Karte verloren hätte? Diese Fragen ergeben sich, solange die Mission nicht deutlich definiert worden ist. Nur mit klaren Angaben, die das endgültige Ziel sowie die genaue Abfolge der Ereignisse definieren, kann der junge 007 seine Mission erfolgreich abschließen. Das beschriebene Beispiel ist eine natürliche Spezifizierung und der Anfangspunkt, von dem aus wir einen Prototyp und eine Simulation erstellen müssen, falls wir eine echtes Kino finden wollen. Dafür ist nun allerdings eine Vielzahl an Lösungsmöglichkeiten denkbar. Weil die Spezifizierung textlich, d. h. nicht-formell ist, ist es unmöglich, zu einem vollständigen Prototypen zu gelangen, der diese Mission erfolgreich beenden könnte.
Beim Design von mobilen Agenten gilt es genau diese Fragen zu berücksichtigen. Wurde die Spezifizierung für die Agenten und deren Aufgaben richtig durchgeführt? Der Terminus "Spezifizierung" wird im Merriam-Webster-Online-Dictionary `http://www.merriam-webster.com` als "a detailed precise presentation of something or of a plan or proposal for something", also als eine genaue und detaillierte Präsentation von etwas oder einem Plan oder einem Vorschlag für etwas" definiert. Wir konzentrieren uns hier sogar auf die "formelle Spezifizierung", denn innerhalb eines mathematischen Konzepts präsentiert, erlaubt sie Beweise oder formelle Analysen und Begründungen.
Die wichtigsten Inhalte unserer Arbeit können wie folgt zusammengefaßt werden:

- sie gibt eine Übersicht über die gegenwärtig in der Entwicklung befindlichen inter-

nationalen Projekte zu "mobilen Agenten" und bespricht die Vor- und Nachteile mobiler Agenten. (Kapitel 2)

- sie beschreibt verschiedene formelle Spezifizierungssprachen für mobile Agenten. (Kapitel 3)
- sie präsentiert und erweitert eine Methode, die formelle Spezifizierungen mit dem Higher Order π-Kalkül schreibt und überprüft Eigenschaften, indem sie Gebrauch von einem model-checker macht. (Kapitel 4)
- sie schlägt drei verschiedene Methoden für die Implementierung eines mobilen Agenten vor und diskutiert ihre Durchführbarkeit, um die angemessenste Methode auszuwählen. (Kapitel 5)
- Nach der Einführung einer theoretischen Basis für die Entwicklung eines mobilen Agenten-Systems entwickeln wir einen Prototypen, genannt *HOPiTool* (Abkürzung steht für "Higher Order Pi-Kalkül Tool"), der die Simulation und Überprüfung von mobilen Agenten vor dem Einsatz innerhalb eines Netzes lokal erlaubt. (Kapitel 6)

Die Standardisierungsherausforderungen

Die Ideen in bezug auf mobile Codes und ihre Implementierung ergaben sich aus der Praxis. Die meisten gegenwärtig verfügbaren und genutzten Standards, Plattformen und Sprachen für die Entwicklung dieser Art von Systemen spiegeln diese Tatsache wieder: sie wurden eher ad hoc konstruiert, als daß sie auf theoretischen Untersuchungen basieren würden. Theoretische rechnerbasierte Modelle sowie entsprechende formelle Spezifizierungssprachen bleiben notwendig, um diese Ideen auf eine solide Basis zu stellen und aufgrund klarer Konzepte das Verstehen, die Analyse und den Vergleich mit anderen Modellen zu ermöglichen. Angefangen mit dem π-Kalkül wurden einige Versuche in Richtung rechnerbedingter Modelle für mobile Systeme unternommen, die z.B. auf abstrakten Zustandmaschinen [16] und mobiler Umgebung gründeten, [23]. Für die praktische Anwendung bedarf es jedoch Spezifizierungssprachen auf höchstem Niveau sowie Programmiersprachen, deren Semantik mittels dieser Systeme beschrieben werden kann. Es gibt einige Vorschläge von entsprechenden Programmiersprachen (z.B. KLAIM [75], Mobile UNITY [66], Pict [86], Nomadic Pict [104]). Aber auf dem Niveau der Spezifizierung und Gültigkeitserklärung/Simulation von mobilen Anwendungen fehlt eine in großem Maße nutzbare formelle Methode, um praktische Systeme zu bauen. In Anbetracht mobiler Codesysteme sind komplexe Verteilungsaspekte wie Lage, Mobilität

und Kommunikation und, in einigen Fällen, Mißerfolge nicht nur Ausführungsprobleme, sondern auch ein Teil der Funktionalität des Systems. In der Obhut von Standardisierungsorganen wie MASIF [28], MASIF FIPA [41], CORBA[85], Corba etc. arbeiten Standardisierungskomitees ständig daran, Standards für Agenten und insbesondere für mobile Agentenprodukte zu erstellen, für die die Hauptlogik in Software umgesetzt werden soll. In den frühen Phasen dieses Prozesses werden viele Eigenschaften, Dienstleistungen, und Funktionalitäten mit Hilfe von informellen operationellen Beschreibungen, Tabellen und Visualisierungen demonstriert. Diese Beschreibungen entwickeln sich dynamisch, und es wird schnell schwierig, mit den Zeichnungen und Gültigkeitserklärungen umzugehen.

Ein praktischer Vorschlag

Der Entwicklungsprozeß von mobilen Agenten, ausgehend von informellen-funktionellen oder operationellen Voraussetzungen bis hin zu formellen Spezifizierungen auf höchster Ebene, ist ein beliebtes Forschungsthema. Jedoch bleiben viele Fragen bestehen, um Voraussetzungen und formelle Spezifizierungen zu definieren, insbesondere bezüglich neuerer Techniken. Formelle Beschreibungstechniken (FDT), wie der Higher Order Pi-Kalkül [90], HOπ wurden geschaffen, um funktionelle Voraussetzungen formell auszudrücken. Insbesondere eignen sie sich sehr gut für die genaue Definition von mobilen Systemen. Auf akademischer wie industrieller Seite zeichnet sich in den letzten Jahren ein starkes Interesse am Design derartiger Szenarios ab. Eine Vielzahl an Notationen wird verwendet, um Designszenarien zu beschreiben: Grammatiken, Automaten, Prozeß-Algebra und Kommunikationsdiagramme.
Mehrere Techniken können für die Lösung von Problemen verwendet werden, die mit den Standardisierungsprozessen und Szenarien verbunden sind. Wir benutzen die HOπ Prozeß-Algebra, um die Spezifizierung von mobilen Agenten-Systemen zu beschreiben. Diese Spezifizierungssprache ist direkt mit den Eigenschaften des Systems verbunden. Um beispielsweise die Leistungseigenschaften in Betracht zu ziehen, ist es wichtig, eine Spezifizierungssprache zu benutzen, die solche Einschränkungen unterstützt. In unserem Fall interessiert die Mobilität dahingehend, daß die Agenten sich richtig von einer Maschine "E" zu einer Maschine "F" bewegen. Die Spezifizierungssprache muß also Deklarationen solcher Bewegungen erlauben. Der Higher Order Pi-Kalkül HOπ gehört zu den Spezifizierungssprachen, die diesen mobilen Aspekt in Betracht zieht.
Nach der vorbereitenden Selektion des geeigneten theoretischen Rahmens und seiner Verbesserung entwickeln wir einen praktischen Prototypen *HOPiTool* für die Validierung und das szenariobasierte Testen, mit dem Ziel, Widersprüchlichkeiten, Am-

biguitäten, Unvollständigkeiten und andere Probleme so bald wie möglich zu erkennen. Ein von den Entwicklern erfordertes praktisches Ergebnis besteht in der Erzeugung einer gültigen Dokumentation, die leicht zu verstehen ist. Außerdem gilt es, eine gültige Testfolge zu erstellen, die in späteren Entwicklungsphasen wiederverwendet werden kann. Wir illustrieren einen Teil dieses Prozesses, indem wir den Dienstlokalisierungssprotokoll (SLP) [52]. als Fallstudie einführen:

Der Vorschlag eines Szenarios für den HOPiTool

Die Verwendung einer formellen Sprache von higher-order wie die HOπ HOπfür einen szenario-basierten Zugang stellt eine vernünftige Entscheidung für die Beschreibung von mobilen und kommunizierenden Systemen dar. Sie lassen sich gut in die Designanordnung einpassen, die wir in Abbildung 2 vorschlagen, wo wir versuchen, die Lücke zwischen informellen Voraussetzungen und dem ersten System-Design zu überbrücken. Anforderungen sind gewöhnlich evolutiv und dynamisch; sie ändern sich und werden

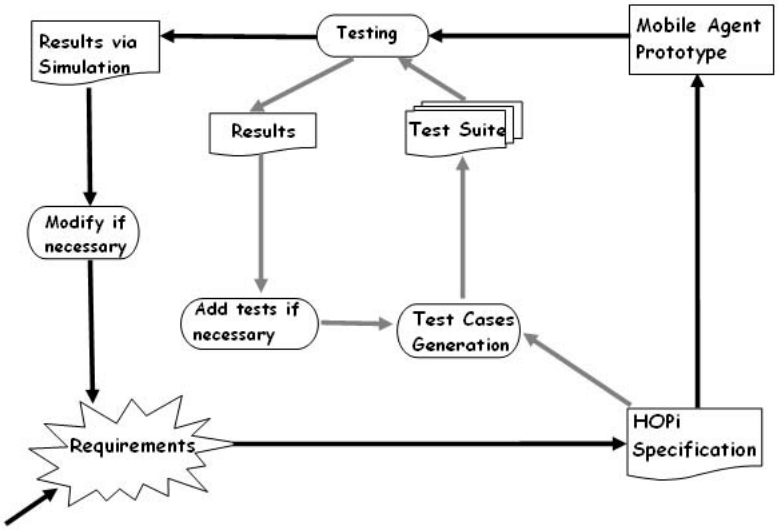


Figure 2: Scenario-Based Approach

von Zeit zu Zeit angepaßt. Deshalb wird ein wiederholender Prozeß das schnelle Prototyping und die direkte Generierung von Testfällen aus Szenarien erlauben. Abbildung 1.1 schlägt eine Methode vor, wo der Hauptzyklus mit der Beschreibung des Szenarios beschäftigt ist. Die higher-order Spezifikation wird die Basis für die Entwicklung unseres Prototyps sein. Gleichzeitig könnten Testdatensätze direkt aus diesem Szenario erzeugt und dann benutzt werden, um den Prototyp hinsichtlich der Spezifizierung

zu überprüfen. Die Ergebnisse, die wir von jenen Tests erhalten, können uns Aufschluß darüber geben, ob für die gewünschte Spezifizierung zusätzliche Testdatensätze nötig sind oder nicht. Um die Eigenschaften des Systems analysieren zu können, ist es notwendig, einen vorhandenen model checker (wie UPPAAL) innerhalb unseres Prototyps einzubetten. Wir nehmen dann an, daß der Prototyp den Anforderungen entspricht. Aus dieser Methode ergaben sich mehrere Vorteile:

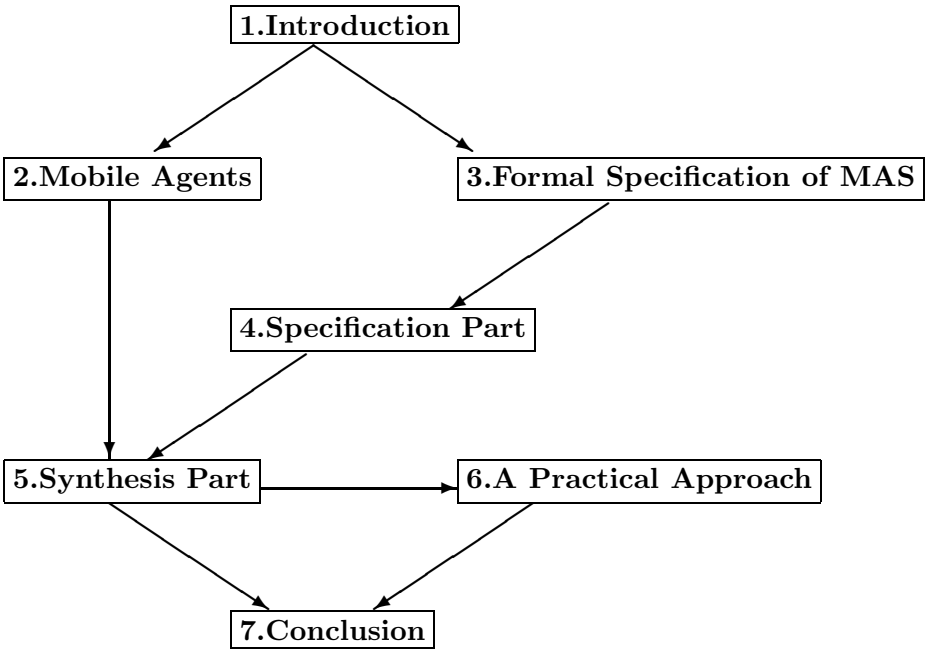
- die Trennung der Funktionalitäten von der zugrundeliegenden Struktur: Szenarien können wieder verwendet werden.
- Schnelles Prototyping: sobald die Struktur und das Szenario ausgewählt und dokumentiert wurden, kann ein Prototyp schnell erzeugt werden.
- Designdokumentation: die Dokumentation wird entlang der Entwicklungsphase erzeugt. Diese Art der Annäherung ermutigt Designer, nützliche Dokumentationen methodisch zu erzeugen.

HOPiTool ist ein Simulationsprototyp fr mobile Agenten, das durch die Abbildung der HOπ-Spezifikation in ein Simulationsmodell sowie durch die automatische Generierung von Code fr eine Mobilittsuntersttzungsplattform mittels Jini/Java, charakterisiert werden kann.

Inhaltsübersicht

Kapitel 2 führt verschiedene mobile Agenten-Systeme sowie mehrere internationale Projekte ein. Die Arbeit bespricht nur einige gegenwärtige Projekte ab dem Jahr 2000. Die Wahl dieser Projekte zeigt die Vielfältigkeit der Anwendungen, die mobile Agenten gebrauchen. Das Thema der mobilen Agenten wird eingeführt, indem verschiedene Interpretationen des Terms "mobiler Agent", verschiedene Anwendungen und Definitionen diskutiert werden. Das folgende Kapitel 3 wird sich auf die verschiedenen Annäherungen, die in der Spezifikation der mobilen Agenten verwendet werden, konzentrieren. Kapitel 4 bespricht eine higher-order formelle Methode für die Mobilität und das Verwenden eines model-checkers, der Systemeigenschaften eines mobilen Systems überprüft. Kapitel 5 schlägt drei verschiedene Methoden vor, wie man ein mobiles Agenten-System am besten durchführt und bespricht die Durchführbarkeit eines praktischen Ergebnisses. Kapitel 6 schlägt eine praktische, auf unsere vorherige theoretische Arbeit begründete, Anwendung vor. Das Prototyp *HOPiTool* wird eingeführt und eine Fallstudie wird besprochen. Kapitel 7 schließt die Arbeit mit einer Zusammenfassung und eine Diskussion über die zukünftigen möglichen Entwicklungen. Die Illustration unten zeigt die

Beziehung zwischen den einzelnen Kapiteln.



Part I

STATE OF THE ART

Chapter 1

Introduction

Today, the rapid development of our information society has lead to an explosive evolution of computer networking. This is particularly obvious in the continuous growth in the number of network users, especially within the Internet. New technologies in the consumer electronics' sector are either offering fast and simple access to the Internet or a means to connect "on-the-go" by way of Global System for Mobile Communication (GSM) networks. Furthermore, the abundance of such services is growing at an evermore accelerated pace.
Associated with this evolution in networking is a trend that is raising more questions: Mobile Computing. Not only laptops are being integrated in computer networks, but also mobile telephones are connecting us to networks via Wireless Application Protocol (WAP) and Wireless Markup Language (WML). Personal organizers, or Personal Data Assistants (PDA's), serve as an additional option for mobile network connectivity. They use techniques of connection such as Infrared Data Association (IrDA), Bluetooth or, like mobile telephones, GSM and Universal Mobile Telecommunications System (UMTS). The applications specified above are operational on not dynamically limited networks, i.e. mobile agents can be added as the execution progresses. The services used by these applications can also use other services,

- whose lifespan is limited in time or providing rights giving a time variable access.
- whose supplier progresses. For example, a service of a phone book can be on a machine "A" and for some problems due to "A", moved to a more progressive machine B.

The important property of these applications is also that the networks are progressive. This development has given rise to many problems and new challenges, of which only a few will be discussed here: the growth of network capacity could not, as yet, cater to the rapid increase in the number of users. Although the Internet has been touted as a network of inexhaustible possibilities, it can only provide a limited range of services and usage without having a permanently connected computer. One way to solve this problem is the recently emerging mobile agents technology.

Mobile agents are programs that are capable of moving (or migrating) from one computer to another within a network and autonomously carrying out commands on behalf of their owner. This migration allows for, both the exploitation of all the member computers' resources within a local network and an efficient execution of a task by means of cooperative problem processing using fellow agents.

For instance, an agent "A" collects a data file C.txt on a machine "C" and then the same agent collects the data file D.txt on a machine called "D". The "locality" characteristic seen here is essential for the improvement of the applications reliability. It is preferable for an agent to move in order to collect data files than reading these remotely. A diminution of networks problems and better security is the advantage of using mobile agents.

If one were to provide the definition of mobile agents using an example in the context of daily life, he/she could apply the following scenario:

Imagine that Bob would like to go to the movies, yet he does not know where his favorite movie is being played tonight. Luckily, Bob has a little brother, Jake, who likes to play agent 007. Jake has now been given the mission of finding out where Bob's favorite movie is shown. To carry out his mission he must go to every cinema in the neighborhood and, if possible, purchase a ticket at the one showing the desired film. After fulfilling the objective, he can then choose to report home on his findings or go play with his other "secret agent" friends. The young 007 feels he has been given a worthy mission and is now in active service. He decides to save the legwork and enlist the help of his agent friends. He calls them up and gives them the assignment of going to the nearest cinema and inquiring about the film. After all have reported in, he collects the findings together and passes on the information to Bob.

Another suitable example, now back in the realm of technology, the Internet and, especially, the World Wide Web (WWW), would be:

A mobile agent has been given the task of retrieving data from any number of data sources (e.g. WWW servers) by sifting through them using a set of defined criteria. The agent would then migrate to the computers and explore their data sources and return the results back to the user instead of merely loading and examining pages containing the various content of the servers over the Internet. According to Theilmann's

thesis [96], this procedure can lead to considerable reductions in, both the strain on the network and the necessary time needed to execute the task, depending on the effort on the part of the agent and the size of the data to be explored and the data being transmitted back to the owner. These advantages coupled together with the ability of agents to carry out their tasks in an asynchronous manner without any further contact to their owners also make them particularly interesting for use with end devices.

Searching and filtering through sources of data are not the only tasks that mobile agents are capable of carrying out. Other uses can be found within the realm of e-commerce, network management, information distribution of parallel computers, services like WebLogic, which use mobile agents in order to find new services on the server and intrusion detection. Various published works have mentioned still other areas of application, such as [72], [46], [94]. The last five to six years have witnessed more and more attempts to apply this technology, yet the applications themselves are negligible when comparing them to the level of the research being conducted today. The "Agents for Remote Action" (Ara) project [84] (1997) is a platform for the portable and secure execution of mobile agents under development at the University of Kaiserslautern. Ara's specific aim in comparison to similar platforms is to provide full mobile agent functionality while retaining as much as possible of established programming models and languages. The projects D'Agents [49] (latest version released in 2002), MadKit [51] from the University of Montpellier (France) and Mole [9] (successfully completed after 5 years in 2000) constitute a few of the research prototypes. The Mole project in Stuttgart was a project in mobile agents (MA) which re-examined MA in the current context and examined such ideas as "MA and reliability", "MA and security" and mixing Java and CORBA within the same system. Other commercially-driven products include Aglets [61] (IBM already provides actual services based on this work, such as TabiCan on the Internet), Grashopper [10] (the agent development platform launched by IKV++ in August 1998, enables the user to create a wealth of applications based on agent technology), LEAP [6] (this project is addressing the need for open infrastructures and services which support dynamic, mobile enterprises) or JADE [11] (Java Agent DEvelopment Framework, a software framework to develop agent-based applications in compliance with the FIPA [39] specifications for interoperable intelligent multi-agent systems). All these examples are very technical platforms, which the programmer needs in order to develop his mobile agents. The notion of mobile agents in these applications is introduced either through the semi-formal and non-formal specification of the system, which contains the notion of mobility, or through a technical library, which is located externally.

Although many technological hurdles have been overcome, such as faster network transfer rates, the use of virtual machines, programming languages that are platform independent (i.e. Java, HTML, XML, PostScript, etc...) and faster, better, more robust

computers, some problems still remain:

Control mechanisms: This area is mainly concerned with the problem of locating and terminating mobile agents. Baumann gives an overview ([8]) of an existing approach, and proposes further solutions for these tasks.

Security: This area of security deals with the protection of both the computer and the mobile agents against attacks from malicious mobile agents and/or implementation environments. F. Hohl has examined this problem in [53].

Fault tolerance: The fault tolerance in the implementation of mobile agents seeks to ensure, among other things, that mobile agents are not blocked or lost due to their interaction with other mobile agents or computers - or in case of network stoppage. A solution for this problem is discussed in [94] by M. Strasser.

Other problems arise from wide-spread geographic distribution, increased network dynamics for dealing with, for example, occurrences of new kinds of services. Another significant problem is the difficulty in testing mobile agents, where, particularly within Internet, it is unclear whether errors encountered are due to the agents themselves or to their environment. Also the test criteria have to be defined concerning the navigation (has an agent gone first through a node C and second through D?) and local activity of an agent (for reading the file $C.txt$ on the node C, does it use the local rights or does it use a remote access?).

1.1 Motivation and Scientific Contribution

The work reported in this thesis is aimed at proposing a technique for modeling and analyzing mobile agent systems, and attempting to bridge the gap between formal mobile agent models and mobile agent implementations.
Looking at the example given about the little brother's task of investigating movie showing times, a few questions could possibly present themselves during the mission. Did Jake, the little brother, understand that the ticket requested was for a show that very evening? Did he have the money to make the purchase? Would he have expressed his brother's desire of buying a ticket to his friends? If so, what were to happen if they did not have enough money? What would have happened had he bought the ticket, yet didn't come home directly deciding, instead, to play with his friends and, in the process, lost the ticket? These questions arise when the mission has not been clearly defined. Only after clearly stating what the definitive objective is and what the exact sequence of

events are, can 007 junior be assured of the end of his mission and, once done, take satisfaction in knowing that his important, secret mission has been successfully completed. The example described above is a natural specification and starting from that we have to design a prototype and carry out a simulation if we want to find a real theater. There are a large number of solutions on how to find a theater. Because the specification is textual i.e. informal, it is impossible to lead to a complete prototype capable to realize this mission.
These are the questions that need to be raised in the design of mobile agents. Has the specification for the agents and their task been correctly made? The word "specification" is defined in the Merriam-Webster Online Dictionary `http//:www.merriam-webster.com` as "a detailed precise presentation of something or of a plan or proposal for something". Here we even focus on "formal specification" i.e., presented within some mathematical framework it allows proofs or formal analysis and reasoning.
The major contributions of our work can be summarized as follows:

- Give an overview of international "mobile agent" projects currently under development and discuss the advantages and disadvantages of mobile agents. (Chapter 2)
- Discuss different formal specification languages in order to specify a mobile agent. (Chapter 3)
- Present and extend an approach where formal specifications are written within the process-algebraic language Higher-Order π-Calculus and also verify some properties using an existing model-checker. (Chapter 4)
- Propose three different methods to implement a mobile agent and discusses their feasibility, in order to choose the most convenient method. (Chapter 5)
- After presenting a theoretical basis which enables the development of a mobile agent system, we develop an prototype called *HOPiTool* (standing for "Higher-Order π-Calculus Tool") that allows simulation and testing of mobile agents locally before executing them within a network. (Chapter 6)

1.1.1 The Standardization Challenges

The ideas around mobile code and its implementations emerged from a practical approach. Most of the standards, platforms and languages currently available and widely used for the development of this kind of systems reflect this fact: they were constructed

in an ad hoc way rather than based on corresponding theoretical investigations. Theoretical computational models as well as corresponding formal specification languages are still necessary to give a sound basis to these ideas, allowing a better understanding (or making concepts clearer), analysis, comparison with other models and even predicting their evolution. Starting with the π-calculus, there had been some efforts towards computational models for mobile systems, e.g. based on abstract state machines [16] and on mobile ambients [23]. However, to be used in practical applications, high-level specification languages as well as programming languages whose semantics can be described using such models must be provided. There are some proposals of corresponding programming languages (e.g. KLAIM [75], Mobile UNITY [66], Pict [86], Nomadic Pict [104]), but on the level of specification and validation/simulation of mobile applications, there is still no formal method that is largely used to build practical systems. When considering mobile code systems, complex distribution aspects, like location and mobility, communication, and, in some cases, failures, are not only implementation issues but also part of the functionality of the system. Under the auspices of standardization bodies such as MASIF [28], FIPA [41], CORBA [85], etc., standardization committees are constantly at work to produce standards for agents and in particular for mobile agents products, for which the main logic is meant to be implemented in software. In the early stages of this processes, many features, services, and functionalities are described using informal operational descriptions, tables and visual. These descriptions evolve dynamically, and their drafting and validation quickly become difficult to manage.
In this context, the following issues should be addressed:

- While designing systems and services at the initial stages, the discussion might focus at a level of detail that is too low with respect to the knowledge (about data, messages, components, etc.) available at the time. Descriptions that include irrelevant details tend to obscure the main idea behind a feature/service/functionality, especially when the latter needs further modifications or refinements. Several levels of detail (abstraction) are often mixed in a single description. For instance, the enumeration of the nodes the agent has to explore can be declared and also the way how to find out the next node (if the roadmap is not known at the beginning of the mission).

- There are possibly ambiguities, inconsistencies or interactions inside or between service descriptions, or between levels of abstraction of a given service. These remain difficult to detect with conventional inspection methods, and often remain hidden until errors are discovered after implementation, at which point correction can be very costly. For instance, in the same case the structure of the messages between agents is more important that the structure of the requests, which are

sent to the host when the agent wants to know which file has to be collected.

1.1.2 A Practical Proposal

The process of going from informal functional or operational requirements to a high-level formal specification is a research subject where much work has been done. However, many challenges still remain, especially regarding newer techniques for defining requirements and formal specifications. Formal Description Techniques (FDT), such as Higher-Order π-calculus (HOπ) [90], were created in order to formally express functional requirements. In particular, they are well suited for the precise definition of mobile systems. Over the last few years, there has been a strong interest, from both academia and industry, in the use of scenarios for system design. However, many different meanings were associated with the word "scenario". They are related to traces (of internal/external events), communication between components, interaction sequences between a system and its user, to a more or less generic collection of such traces, etc. Numerous notations are also used to describe them: grammars, automata, process algebras and communication diagrams.
Several techniques can be used to address the issues related to standardization processes and scenarios. We use the HOπ process algebra to describe the specification of mobile agent systems. The specification language (or rather the expression strength of the specification language) is directly linked to the definition of the system properties which one wishes to establish. For example, to take into account performance properties, it is essential to use a specification language that supports these constraints. In our case, one is interested in the mobility and wants to establish properties stating that the agents move correctly from a machine "E" to a machine "F", then it is necessary that the specification language allows declarations of such movements. The Higher-Order π-Calculus is such a specification language, which takes into account the mobility aspect.
After the preliminary work of selecting and improving the most adequate theoretical framework, we develop a practical prototype *HOPiTool* for the validation and for the scenario based testing, in order to detect inconsistencies, ambiguities, incompleteness, and other problems as soon as possible. One of the practical results, needed by the developers, is to produce documentation that is more easily understood and also is validated as much as possible, and to produce a validated test suite that can be reused at later stages, including implementation. The π-Calculus is hidden within the core of the test suite. We illustrate part of this process on a case study involving the Service Location Protocol (SLP) [52].

1.1.3 A Proposed Scenario for the HOPiTool

The use of a higher-order formal language as the HOπ in a scenario-based approach represents a judicious choice for the description of mobile and communicating systems. They fit well in the design approach that we propose in Figure 1.1, where we intend to bridge the gap between informal requirements and the first system design. Requirements

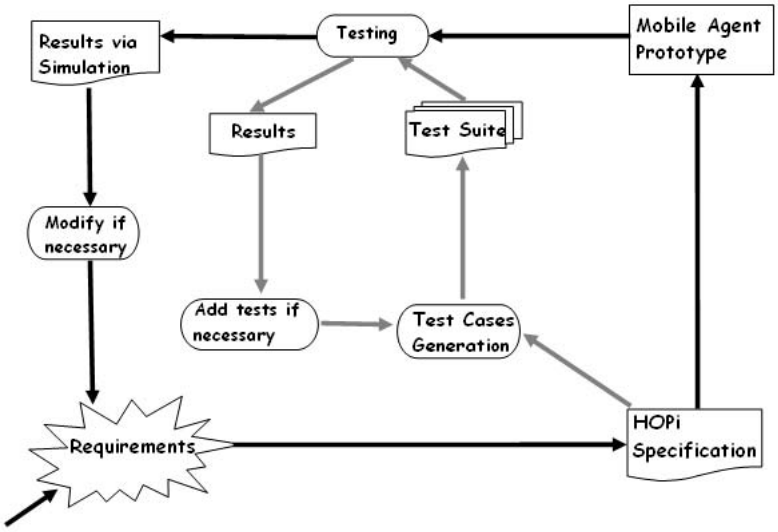


Figure 1.1: Scenario-Based Approach

are usually evolutive and dynamic; they change and are adapted over time. This is why an iterative and incremental process will allow rapid prototyping and test cases generation directly from scenarios. Figure 1.1 proposes an approach where the main cycle is concerned with the description of the scenarios. They are then merged in order to (manually) synthesize a higher-order specification, which will be the basis for the construction of our prototype. Concurrently, test cases could be generated from these scenarios and then be used to test the prototype with respect to the specification. The results obtained from those tests will allow us to see whether or not additional test cases are necessary in order to achieve the desired specification coverage. In order to verify and analyze properties of our system it is necessary to embed an existing model checker (like UPPAAL) [12] within our prototype. We then consider that the prototype meets the requirements. This approach presents several advantages:

- Separation of the functionalities from the underlying structure: since scenarios are formalized at a higher level of abstraction than message exchanges, different

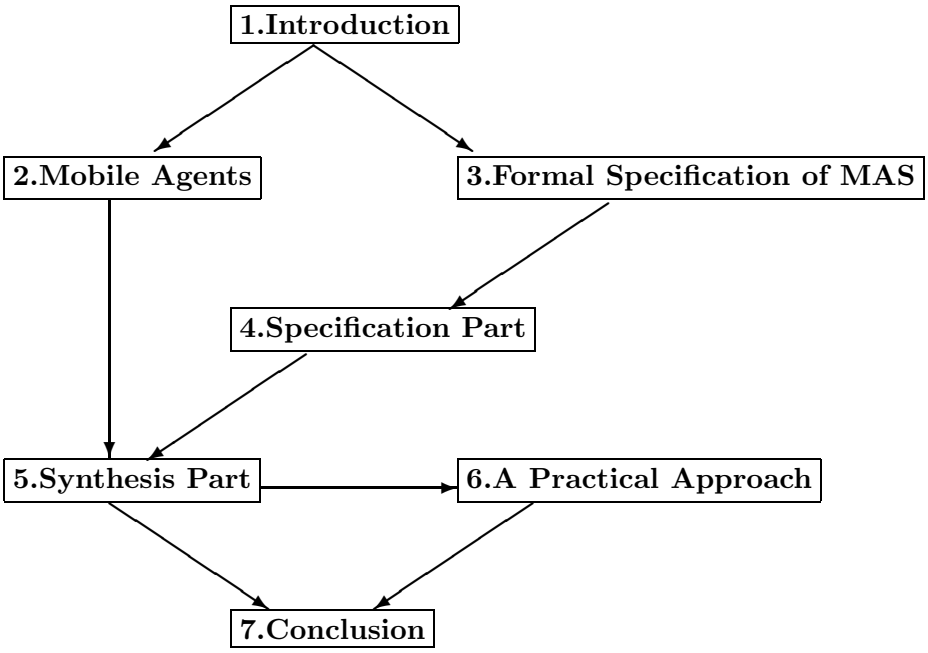
underlying structures or architectures can be evaluated with more flexibility. The scenarios then become highly reusable entities. As mentioned below, they can be used again to test the implementation.

- Fast prototyping: once the structure and the scenarios are selected and documented, a prototype can then be generated rapidly.
- Design documentation: the documentation is done as we go along the design cycle. Very often, designers document their design only when they have to; this kind of approach encourages designers to methodically produce useful documentation.

Applications of mobile agents, written according to the formal language HOπ and the approach shown before, can be simulated with the simulation prototype *HOPiTool*. Moreover, code can be automatically generated following a straightforward mapping to Java classes considering the use of a mobility support platform, such as Java Intelligent Network Infrastructure (Jini)[77].
The *HOPiTool* is a prototype for the simulation of mobile agents and can be characterized by the mapping of a higher-order π-Calculus specification into a simulation model and the generation of code for a mobility support platform.

1.2 Outline

Chapter 2 introduces different mobile agent systems and several worldwide projects based on these. The work discusses only some current projects from the year 2000 to present. The choice of these projects shows the variety of applications using mobile agents. The topic of mobile agents has been introduced by covering the various interpretations of the term "mobile agents", outlining the possible applications of mobile agents and presenting the definition of mobile agents as used within this work. Then chapter 3 focuses on the different approaches used in the specification of mobile agents. Chapter 4 discusses a higher-order formal method for mobility and applying a model checker that verifies some system properties of a mobility protocol. Chapter 5 proposes three different methods on how to implement a mobile agent system and discusses the feasibility of a practical result. Chapter 6 proposes a practical application based on our previous theoretical work. The prototype *HOPiTool* will be introduced and a case study will be discussed. Chapter 7 concludes the work with a summary, a discussion on transferring the desired results to other areas and an outlook on future developments. The illustration below shows the chapters as well as the relations between each other.


1.Introduction
2.Mobile Agents
3.Formal Specification of MAS
4.Specification Part
5.Synthesis Part
6.A Practical Approach
7.Conclusion

Chapter 2

Mobile Agents

A 600 foot waterfall whose face is frozen solid. Pushing in still closer, there is a tiny black dot inching its way up the ice. A human figure. This is: `JAMES BOND, BRITISH SECRET SERVICE AGENT, 007` *[93]. Bond is sweating and straining, four hundred feet in the air. He has an ice-pick tethered to each hand, ice-cleats on his boots. A black backpack. Bond climbs. Huffing, sweating, he goes up, and up, until another angle. Bond pauses. Thinks.*

James Bond is a human agent, he can move around the world, he can carry things, he is autonomous, he always performs some jobs on behalf of his employer and the most important thing is that he thinks. Like in many others scientific areas, the humanity progresses by taking examples from his environments and nature. The human agents were the model of a software agent and researchers over the world are trying to perfect the software, to make it "human". The ideal of a software agent is one, he can act in the same way a human agent does. Nowadays the software agents are not "human", they use artificial intelligence in stand of thinking and they can move around the network, are autonomous and carry out some jobs. Mobile agents systems are special software agents, which provide a particular program paradigm.
This chapter examines the state-of-the-art enabling technologies for mobile agents, where the term of the mobile agent system will be explained from a description of the term "mobile code system". The first part lists a number of potential advantages and disadvantages of mobile agents concluding with several recent success stories in this field. The second part of this chapter deals with code mobility, existing mobility forms, design patterns and standardizations. The chapter concludes with some personal reflections about the usefulness of the research on "mobile agents" nowadays and explains that our research brings solutions to cover some of the shown disadvantages of mobile agents.

2.1 Why Mobile Agents?

Mobile agents are software abstractions that can migrate across the network representing users in various tasks. This is a contentious topic [71] that attracts some researchers and repels others. Some dislike the "mobile" attribute; the others the "agent" noun. Mobile agent opponents believe that most problems addressed by mobility can be equally well, yet more easily and more securely, solved by static clients that exchange messages. Those who favor mobility justify its advantage over static alternatives with benefits, such as improved locality of reference, ability to represent disconnected users, flexibility, and customization. The ideas of mobile abstractions are probably as old as distributed systems.

Mobility is an orthogonal property of agents, i.e., not all agents are mobile. An agent can just sit there and communicate with its environment through conventional means, such as remote procedure calling and messaging. Agents that do not or cannot move are called stationary agents. A stationary agent executes only on the system on which it starts executing. If it needs information on another system or needs to interact with an agent on another system, it typically uses a communication mechanism, such as remote procedure calling. In contrast, a mobile agent is not bound to the system on which its execution starts [60]. It is free to travel among the hosts in the network. Created in one execution environment, it can transport its "state" and "code" with it to another execution environment in the network, where it resumes execution. The term "state" typically means the attribute values of the agent that help it to determine what to do when it resumes execution at its destination. "Code" in an object-oriented context means the class code necessary for an agent to execute. A mobile agent has the unique ability to transport itself from one system in a network to another in the same network. This ability allows it to move to a system containing an agent with which it wants to interact and then to take advantage of being in the same host or network as the agent. To make use of mobile agents, a system has to incorporate a mobility framework. The framework has to provide facilities that support all of the agent models, including the navigation model.

For the life-cycle model, services to create, destroy, start, suspend, stop, etc., agents are needed.

The computational model refers to the computational capabilities of an agent, which include data manipulation and thread control primitives.

The security model describes the ways in which agents can access network resources, as well as the ways of accessing the internals of the agents from the network.

The communication model defines the communication between agents and between an agent and other entities (e.g., the network or database).

The navigation model All issues referring to transporting an agent (with or without its state) between two computational entities residing in different locations are handled by the navigation model.

Obviously, the framework incurs certain costs including increased memory requirements and execution and access delays on every participating device. The underlying technology, however, is evolving rapidly. For example, the footprints of certain Java Virtual Machines (JVM), which are the basis for many mobile agent frameworks, are very small making them suitable for embedded systems [58]. Some researchers like in [14] believe that the use of Java chips will be important in the future networked devices. In addition, forthcoming new software packages like Jini [59] address many of the needs of agent-based systems. The size of mobile agents depends on what they do. In swarm intelligence [103], the agents are very small. On the other hand, configuration or diagnostic agents might get quite big, because they need to encode complex algorithms or reasoning engines. However, agents can extend their capabilities on-the-fly, on-site by downloading required code off the network. They can carry only the minimum functionality, which can grow depending on the local environment and needs. This capability is facilitated by code mobility.

2.1.1 Reasons for Success

According to [62] and [36] there are many areas that may benefit from appropriate use of mobile agents instead of, or in addition to, classical client/server models.

1. *Mobile agents decrease the network load.*
   Concurrent and distributed systems often rely on communication protocols engaging multiple interactions to perform a given task. The result is a lot of network traffic. Mobile agents allow users to package a conversation and dispatch it to a destination host where interactions take place locally. Mobile agents are also useful when reducing the flow of raw data in the network. When very large volumes of data are stored at remote hosts, that data should be processed in its locality rather than transferred over the network. The motto for agent-based data processing is simple: Move the computation to the data rather than the data to the computation.

2. *Mobile agents overcome network latency.*

Critical real-time systems, such as robots in manufacturing processes, need to respond in real time to changes in their environments. Controlling such systems through a factory network of substantial size involves significant latencies. For critical real-time systems, such latencies are not acceptable. Mobile agents offer a solution, because they can be dispatched from a central controller to act locally and execute the controller's directions directly.

3. *Mobile agents encapsulate protocols.*
   When data is exchanged in a distributed system, each host owns the code that implements the protocols needed to properly encode outgoing data and interpret incoming data. However, as protocols evolve to accommodate new requirements for efficiency or security, it is cumbersome if not impossible to upgrade protocol code properly. As a result, protocols often become a legacy problem. Mobile agents, on the other hand, can move to remote hosts to establish "channels" based on proprietary protocols.

4. *Mobile agents perform asynchronously and autonomously.*
   Mobile devices often rely on expensive or fragile network connections. Tasks requiring a continuously open connection between a mobile device and a fixed network are probably not economically or technically feasible. To solve this problem, tasks can be embedded into mobile agents, which can then be dispatched into the network. After being dispatched, the agents become independent of the process that created them and can operate asynchronously and autonomously. Mobile agents can be delegated to perform certain tasks even if the delegating entity does not remain active.

5. *Mobile agents adapt dynamically.*
   Mobile agents can sense their execution environment and react autonomously to changes. Multiple mobile agents have the unique ability of distributing themselves among the hosts in the network to maintain the optimal configuration for solving a particular problem.

6. *Mobile agents support heterogeneous environments.*
   Mobile agents are separated from the hosts by the mobility framework. If the framework is in place, agents can target any system. The costs of running a Java Virtual Machine (JVM) on a device are decreasing. Java chips will probably dominate in the future, but the underlying technology is also evolving in the direction of ever-smaller footprints (e.g., Jini [59]).

7. *Mobile agents are robust and fault-tolerant.*

Mobile agents' ability to react dynamically to unfavorable situations and events makes it easier to build robust and fault-tolerant distributed systems. If a host is being shut down, all agents executing on that machine are warned and given time to dispatch and continue their operation on another host in the network.

8. *Mobile agents generate savings in efficiency.*
   CPU consumption is limited, because a mobile agent executes only on one node at a time. Other nodes do not run an agent until needed.

9. *Mobile agents require less space.*
   Resource consumption is limited, because a mobile agent resides only on one node at a time. In contrast, static multiple servers require duplication of functionality at every location. Mobile agents carry the functionality with them, so it does not have to be duplicated. Remote objects provide similar benefits, but the costs of the middleware might be high.

10. *Mobile agents extend online services.*
    Mobile agents can be used to extend capabilities of applications, for example, providing services. This allows for building systems that are extremely flexible.

11. *Mobile agents facilitate software upgrades.*
    A mobile agent can be exchanged virtually at will. In contrast, swapping functionality of servers is complicated; especially, if we want to maintain the appropriate level of quality of service (QoS).

One can say that mobile agents will most likely be useful in three general areas:

- One is disconnected computing, such as laptops and PDAs, because they frequently disconnect from the network or use a wireless network that might become disconnected on short notice.
- The second is information retrieval situations, like applications where the agent can be sent to a large data source and filter through the data locally.
- The third category is dynamic deployment of software.

Imagine a large organization has hundreds of PDAs in its workforce, and they all need to be reconfigured with a new software version or some data set. A mobile agent can convey a new file to everybody's PDA. But, if it involves some reconfiguration, the code represented by the mobile agent is useful.
However, a killer application for mobile agents does not exist yet, because almost everything one can do with mobile agents could be done with some other, more traditional

technology (like client to server applications, RMI, etc.). The mobile agents technology can solve a lot of problems in a uniform way rather than a technology that enables completely new things that were not possible another way. Several applications [62] clearly benefit from the mobile agent paradigm:

- *E-commerce.*
  Mobile agents are well suited for e-commerce. A commercial transaction may require real-time access to remote resources, such as stock quotes and perhaps even agent-to-agent negotiation. Different agents have different goals and implement and exercise different strategies to accomplish them. We envision agents embodying the intentions of their creators, acting and negotiating on their behalf. Mobile agent technology is a very appealing solution for this kind of problem.

- *Personal assistance.*
  Mobile agents' ability to execute on remote hosts makes them suitable as assistants performing tasks in the network on behalf of their creators. Remote assistants operate independently of their limited network connectivity. For example, to schedule a meeting with several other people, a user can send a mobile agent to interact with the agents representing each of the people invited to the meeting. The agents negotiate and establish a meeting time.

- *Secure brokering.*
  An interesting application of mobile agents is in collaborations in which not all collaborators are trusted. The parties could let their mobile agents meet on a mutually agreed secure host where collaboration takes place without risk of the host taking the side of one of the visiting agents.

- *Distributed information retrieval.*
  Instead of moving large amounts of data to the search engine so it can create search indexes, agent creators can dispatch their agents to remote information sources where they locally create search indexes that can later be shipped back to the system of origin. Mobile agents can also perform extended searches that are not constrained by the hours during which a creator's computer is operational.

- *Telecommunication networks services.*
  Support and management of advanced telecommunication services are characterized by dynamic network reconfiguration and user customization. The physical size of these networks and the strict requirements under which they operate call for mobile agent technology to function as the glue keeping the systems flexible yet effective.

- *Workflow applications and groupware.*
  The nature of workflow applications includes support for the flow of information among coworkers. Mobile agents are especially useful here, because, in addition to mobility, they provide a degree of autonomy to the workflow item. Also they represent a method to insure the persistence of a property over a host group. Individual workflow items fully embody the information and behavior they need to move through the organization - independent of any particular application.

- *Monitoring and notification.*
  This classic mobile agent application highlights the asynchronous nature of these agents. An agent can monitor a given information source without being dependent on the system from which it originates. Agents can be dispatched to wait for certain kinds of information to become available. It is often important that the life spans of monitoring agents exceed or be independent of the computing processes that created them.

- *Information distribution.*
  Mobile agents embody the so-called Internet push model. Agents can distribute information, such as news, automatic software updates and administration of licences for vendors. The agents bring the new software components, as well as installation procedures, directly to customers' computers where they autonomously update and manage the software.

- *Parallel processing.*
  It is a method to distribute the processing of data. Given that mobile agents can create a cascade of clones in the network, another potential use of mobile agent technology is to administer parallel processing tasks. If a computation requires so much processor power that it must be distributed among multiple processors, an infrastructure of mobile agent hosts can be a plausible way to allocate the related processes.

2.1.2 Reasons for Downfall

Mobile agents provide a very appealing, intuitive, and apparently simple abstraction. Unfortunately there are many difficult problems that have to be addressed in order to make mobile agent-based applications work reliably. After an enormous enthusiasm referred to mobile agents in the past ten years, new beliefs and opinions are emerging. They are some justified worries about the mobile agents but naysayers claim that the "only widespread incarnation of mobile software agents is malware" [87].

The biggest issue concerning the mobile agents remains the security aspect. For mobile systems to support security means to consider several problems:

- Servers are exposed to the risk of system penetration by malicious agents, which may leak sensitive information.
- Sensitive data contained within an agent (such as its user's credit card number, personal preferences, etc.) may be compromised, due to eavesdropping on insecure networks, or if the agent executes on a malicious server.
- The agent's code, control flow and results could be altered by servers for malicious purposes.
- Agents may mount "denial of service" attacks on servers, whereby they hog server resources and prevent other agents from progressing.
- The name service database could be subjected to tampering. For example, the public keys of various principals could be modified, or malicious users may create names in other users' namespaces.

According to [100] there are ten reasons for failure of mobile agents.

1. *Mobile agents do not perform well* because in the general case, they provide worse performance than other mechanisms, such as remote evaluation and they are very expansive.
2. *Mobile agents are difficult to design* because it is hard to clearly identify which components will be interacting and how this interaction can be modeled.
3. *Mobile agents are difficult to develop* because the development of a mobile agent-based application is a daunting task. The code has to be implemented so that it runs in unpredictable environments, possibly interfacing with other mobile agents and static components never seen before. Being able to foresee the type of operating environment the mobile agent will be running in is very difficult. In addition, the technologies that should support the development process are often just prototypes that do not provide the type of support needed to develop real-world applications.
4. *Mobile agents are difficult to test and debug* because distribution and mobility from one node to another in a non pre-defined order add complexity to this process.

5. *Mobile agents are difficult to authenticate and control* because it is not clear which identities should be authenticated and how the access control mechanisms should take into account this information.

6. *Mobile agents can be "brainwashed"* because they are vulnerable to attacks coming from malicious hosts, when they travel across multiple hosts to complete their tasks. For example, a malicious host can modify the code or memory image of an agent to change the way the agent behaves. The result of this "brainwashing" attack would be the creation of a malicious agent whose actions will be attributed to one of the identities initially associated with the agent. This type of attack is extremely difficult (if not impossible) to detect and prevent.

7. *Mobile agents cannot keep secrets* because they have been advocated as a means to implement e-commerce and other critical applications. To perform sensitive transactions (e.g., signing a contract), it is often necessary to perform operations that require secret material, such as a private key. Unfortunately, a secret cannot be effectively concealed if it has to be used by an agent on a remote host and the agent cannot interact with the "home base". Even though some solutions based on the evaluation of encrypted functions have been proposed in [88], no practical application of this mechanism has been devised.

8. *Mobile agents lack a ubiquitous infrastructure* because they require an infrastructure that supports the tasks of marshaling, transfer, and unmarshaling an agent's representation. This infrastructure needs to be deployed on every host that can possibly be the recipient of an agent. This is a requirement that is difficult to meet especially because existing infrastructures have been proven to be vulnerable to a number of attacks [40].

9. *Mobile agents lack a shared language/ontology* because they need to interact with the environment they visit in order to achieve their goals. This interaction requires that the format used to exchange data and the meaning that is associated with the data is understood and agreed upon by both the agent and the partner of the interaction. Even though a number of these shared languages/ontologies have been proposed, there is still no widely accepted language or ontology.

10. *Mobile agents are suspiciously similar to worms* because they are components that autonomously trigger the transfer of their image to a remote host where they restart execution. This mechanism has some striking similarity to the way malicious worms spread across networks. Worms have proven to be extremely difficult to eradicate and some worms are nowadays considered part of the "background

noise" of the Internet. A mobile agent infrastructure would support the execution of both benign and malicious agents and, therefore, it would be prone to be leveraged to launch worm-like attacks.

The conclusion of [100] expresses the conviction of the author, that mobile agents did not meet the expectation they raised ten years ago in terms of widespread deployment and use; and advocates the use of other forms of mobility, like remote evaluation or code on demand.
Other authors conclude in their articles [87] and [15] that the use of Java for mobile applications was the wrong choice and led to the building of malicious software (malware). Java makes it impossible to build and maintain a publicly deployed and dependable mobile agent system. Several deficiencies are given:

- lack of application separation; this is addressed in JSR 121 [24], but respective extensions will not likely be included before JDK Version 1.6 (the current version is 1.4);
- there is no safe method to force a Java thread to stop; adverse code may easily catch any exceptions pertaining to its elimination;
- the Garbage Collector thread may be hijacked by directly or indirectly overriding finalization methods;
- adverse code may block on globally visible class locks, thereby locking other threads vital to the functioning of the runtime system;
- the security model is flexible but access control checks are dispersed throughout the installed classes; a single unguarded privileged action implementation easily undermines the security of the virtual machine.

2.1.3 Selected Success Stories

This section lists some mobile agents projects, who in spite of the criticism of the last two years, successfully ended. The tables 2.1 and 2.2 show selected university research and industrial projects around the world. This list does not claim to be complete, the used selection criteria among other things were: the project end not older than three years, a highly public interest on the project, a variation of mobile agent platform used in this projects and an internationalization (research from the USA, Germany, France, Pakistan and the European Commission).

Projects Mobile Agents	**Country**	**Project Length**	**State**	**Mobile Agent Platform used**	**Institution**
ActComm Project	USA	1997-2002	successfully completed	D'Agents	Dartmouth College and MURI
AMASE	Germany	1998-	open	own platform	Deutsche Telekom Berkom GmbH
CoABS	USA	1998-	open	Java-Jini	Air Force Research Laboratory DARPA
CogVis	Germany	2001-2005	open	Mobile agent software	Information Society Technologie
DIAMOnDS	Pakistan	2002-2003	successfully completed	Jini 1.1	NIIT University of Islamabad
Dilema	IST European Commission	2000-2002	successfully completed	LANA	European Industrial project
HAWK	Germany	1998-2000	closed	Java	University of Stuttgart
LEAP	France	2000-2002	successfully completed	JADE	Motorola France

Table 2.1: Overview I: Projects using Mobile Agents

MadKit	France	2002-	open	Java, Scheme, Jess	University of Montpellier, France
MANTRIP	IST European Commission	2000-2002	completed	MAT	Solinet Germany
MAP	Germany	1998-2003	successfully completed	SeMoA	DLR - German Center for Air and Space-Travel, Siemens
MOJAVE	USA	2001-2004	successfully completed	Jini	Motorola Labs
SysteMATech	Germany	1999-2005	open	Mobile agent software	iVS , DFS, accis
TeleCare	IST European Commission	2001-2004	successfully completed	own platform	UNINOVA Uni Amsterdam, Synchronix, Skill, Camara, RoundRose

Table 2.2: Overview II: Projects using Mobile Agents

The **ActComm** : "Project on Transportable Agents and Wireless Networks" [48] is funded by the "Air Force Office Of Scientific Research" through a "Department of Defence Multidisciplinary University Research Initiative" (MURI) grant.
The goal was to develop technologies that will maximize the usability of complex, global computer and communications networks, focusing especially on wireless networks, for modern command-and-control applications.
The technical innovations are: active software, active information, active hybrid networks and active resource allocation. The latest version of D'Agents is incorporated into the demo.
Future military wireless computer and communications networks will be more robust, more powerful and more flexible under a wide variety of operating environments. Active elements are coordinated by a novel architecture that uses advanced agents to manage network, computer and information assets delivering high confidence communications

and computing.
AMASE : "A Complete Agent Platform for the Wireless Mobile Communication Environment" [83].
AMASE investigates how to enhance existing agent platforms in order to support stationary and mobile agents for the wireless mobile communication environment, and how to enable agent based mobile access to Multimedia Information Services.
Objectives are to:

- Identify and develop techniques for agent technology to be efficient and effective for wireless mobile applications;
- Benchmark stationary and mobile agent techniques in the wireless environment to understand their efficiency in cost and response time under different operating conditions;
- Demonstrate how agent technology helps to create customizable and adaptive applications in the wireless mobile environment; and
- Influence and/or adopt the development of agent-related standards.

The AMASE project develops a generic open communication platform on its own technology.
CoABS : "Control of Agent-Based Systems" [19].
The Control of Agent-Based Systems (CoABS) program develops and evaluates a wide variety of alternative agent control and coordination strategies to determine the most effective strategies for achieving the benefits of agent-based systems, while insuring that self-organizing agent systems will maintain acceptable performance and security protections.
The CoABS program develops and evaluates several control strategies that will allow military commanders and planners to automate relevant command and control functions such as information gathering and filtering, mission planning and execution monitoring, and information system protection. Through the effective control of agent systems, the intelligent agents will work in harmony to strengthen significantly military capability by reducing planning time, automating and protecting Command and Control functions, and enhancing decision-making.
CogVis : "Cognitive Vision Systems" [26].
The objective of this project is to develop methods and techniques that enable to construct truly cognitive vision systems. In the context of an embodied agent such vision systems should be able to perform categorization and recognition of objects and events

in a task-oriented manner and in realistic environments. The functionality will for example enable construction of mobile agents that can interpret the action of humans and interact with the environment for tasks such as fetch and delivery of objects in a realistic domestic setting. The cognitive capabilities of a vision system can be greatly enhanced by providing it with a memory of past visual experiences, a vision memory. As the diverse uses of visual experience indicate, a vision memory may by no means be restricted to the storage of only one particular level of representation. Rather, visual experiences should describe visual activities as completely as useful and manageable, as a multi-level and multi-modal pattern. For example, the visual experience of a mobile agent should associate visual phenomena with attentional and task level information, ego-motion, relevant spatial and temporal context, etc. The design of a vision memory is one of the major tasks of this project.

DIAMOnDS: "Distributed Agents for Mobile and Dynamic Services." [92]

This project provides a secure and flexible distributed services management infrastructure which can be used for communications and coupling of distributed services used in CERN CMS community. The main features of the framework application are the presentation of agents and their hosts as services; it also allows remote monitoring of agents by providing remotely downloadable GUI and allows agents to be accessed from the web.

The prototype system DIAMOnDS, that has been developed, allows a service to send agents on its behalf to other services to perform data manipulation and processing. Remote file system access functionality has been incorporated by the agent framework and allows services to dynamically share and browse the file system resources of hosts running the services. A generic data base access functionality was also implemented in the mobile agent framework allowing performing efficiently complex data mining and processing operations in distributed systems. Agents have been implemented as mobile services that are discovered using the Jini Service Lookup mechanism and used by other services for task management and communication. Agents provide complex proxies for the interaction with other services as well as specific GUI to monitor and control the agent activity. Thus agents acting on behalf of one service cooperate with other services to carry out a job, providing inter-operation of loosely coupled services in semi-autonomous way.

DILEMMA : "Digital Design and Life-cycle Management for Distributed Information Supply Services in Innovation Exploitation and Technology Transfer" [30].

The aim of the DILEMMA project was to design and develop an integrated system that would provide the basis for the efficient technology transfer and innovation exploitation between European enterprizes, research organizations, experts and public or private organizations, with the assistance of mediators (brokers) by facilitating and improving

information supply and communication between them. The DILEMMA system encompasses features expected to dominate in distributed service provision environments, which adopt a decentralized approach in all aspects related to the service design, provision and the lifecycle management. A DILEMMA service is formed by fundamental service elements, which are implemented by means of Mobile Agents technology. The implementation is platform-independent and uses common components. A mobile agent infrastructure, called LANA, developed by the Object Systems Group of the University of Geneva is used to route DILEMMA service elements intelligently, and to carry out other key information supply-related operations. The DILEMMA agent infrastructure thus supports the decentralized process execution that is so essential to the dynamic character of an efficiently networked inter-IRC organization.
The project **Hawk** : "Harvesting the Widely Distributed Knowledge" [97] aims at developing new tools for searching the Internet such that precise and comprehensive searching becomes feasible in a scalable and efficient way.
So far, the project consists of three subprojects which constitute a hierarchy of solutions and which are concerned with:

- Search engines that are specialized for single domains.
- Dissemination of mobile agent / mobile code for performing the task of distributed information filtering much more efficient than it is possible with traditional client-server techniques.
- Distance Maps of the Internet, that allow estimating the network distance between arbitrary network hosts.

These subprojects make use of each other, i.e. the specialized search engines use mobile code technology for realizing an efficient resource access and the algorithms for the optimal coordination of mobile filter programs exploit the network distance knowledge provided by the distance maps. However, each of these solutions can be also used independently of the others and in many different contexts.
LEAP : "Lightweight Extensible Agent Platform" [6].
The LEAP project is addressing the need for open infrastructures and services which support dynamic, mobile enterprizes. It developed agent-based services supporting three requirements of a mobile enterprize workforce: Knowledge management (anticipating individual knowledge requirements), decentralized work co-ordination (empowering individuals, co-ordinating and trading jobs) and travel management (planning and coordinating individual travel needs).
Central to these agent-based services is the need for a standardized Agent Platform. The

LEAP project developed an agent platform that is: lightweight, executable on small devices such as PDAs and phones; extensible, in size and functionality; operating system diagnostic; mobile team management application enabling, supporting wired and wireless communications and FIPA compliant.

MadKit [51] is a modular and scalable multi-agent platform written in Java and built upon the AGR (Agent/Group/Role) organizational model: agents are situated in groups and play roles. MadKit allows high heterogeneity in agent architectures and communication languages, and various customizations. MadKit communication is based on a peer-to-peer mechanism, and allows developers to quickly develop distributed applications using multi-agent principles. Agents in MadKit may be programmed in Java, Scheme (Kawa), Jess (rule based engine) or BeanShell. Other script languages may be easily added. MadKit comes with a full set of facilities and agents for launching, displaying, developing and monitoring agents and organizations. MadKit is a free software based on the GPL/LGPL license.

MANTRIP : "MANagement Testing and Reconfiguration of IP based networks using mobile software agents" [74].

The main goal of the project is to design, develop, test, validate and provide a set of novel network management applications based on Mobile Agent Technology (MAT) for managing IP based networks and to evaluate MAT in the context of Network Management. More specifically the project produces three network management applications based on MAT technology:

- An application for Configuration and Alarm management of systems for the access network.
- An application for configuring QoS parameters within an IP based administrative domain and accordingly monitoring / auditing the delivered QoS. The QoS parameters are considered part of a Service Level Agreement (SLA) with a neighboring domain (customer or ISP).
- An application for Conformance Testing of network elements and mobile agents as well as for monitoring signalling and mobile agents' internal behaviour.

MAP : "Mobile Adaptive Procedure" [102].

The MAP project will develop a system supporting civil servants while they interact with citizens; it is a front-end e-assistance system. Special software agents will "listen" to the interaction in real-time, using advanced speech-recognition technologies, to identify the topics discussed. An expert system will select from the knowledge base of the administration the most pertinent information; then MAP will proactively propose it to

the civil servant. The system will assist the interaction with the citizens in a seamless way, whenever they contact the administration: while on the move, on the net, face-to-face.

Mojave : "MObile Jini Agent enVironmEnt" [98] is an agent-based platform for building and deploying "malleable" services, i.e., services that are environment-aware and that can adapt to changing environment conditions either by migrating to a more suitable environment or by reconfiguring themselves to provide the best service possible under the new conditions. The concept of malleable services lends itself particularly well to applications that must operate, and survive, in rapidly changing environments. Ideal candidates include network systems management and intrusion-tolerant systems. Mobile agents with their inherent capacity for mobility and autonomous behavior – provide the ideal building block for such malleable services. Such agents must be programmable, and require the services of a "container" to host the agents and to provide the housekeeping services (e.g., create, clone, dispose, or move agents) and support services (e.g., communication, event monitoring) required for agent operation. The Mojave system satisfies both these requirements. It provides a simple set of APIs and utility classes that can be exploited to develop custom agent applications. It also provides an infrastructure to support these applications in a robust and extensible manner. The Mojave infrastructure consists of three components (pods, liaisons and agents), and exploits two different distributed computing concepts: Jini (distributed computing) and tuples-paces (distributed shared memory).

The **SysteMATech** : "System Management based on Mobile Agent Technology" [95]. SysteMATech is a project that aims to maximize the application of system and network management platforms based on a prototype tool known as the IntraManager. This prototype uses mobile agents to decentralize network management and equips these agents with the ability to forecast system states. The result is a management solution designed to solve some of the common problems in complex networks. The premise is the value of collaboration, where academia and industry combine forces - this expertise is bringing research and industry closer together. With the system management market dominated by expensive, high maintenance solutions, Systematech aims at filling a gap in providing an effective solution to small to mid-sized companies, as well as to establishing a European voice in the System Management marketplace.

Partners in this project are: the "accsis GmbH" [`http://www.accsis.de`] which focuses on the exploitation of state-of-the-art information technology know-how and the transfer of technology from research to practice; the "DFS Deutsche Flugsicherung GmbH" [`http://www.dfs.de`] which provides air navigation services to the Federal Republic of Germany; and the "iVS - "Intelligente Netze und Management verteilter Systeme" [`http://www.ivs.tu-berlin.de/`] that is a research group for intelligent networks and

management of distributed systems management at the Technical University of Berlin. The overall goal in **TeleCARE** [21] is the design and development of a configurable framework and new technological solutions for tele-supervision and tele-assistance, based on the integration of a multi-agent and a federated information management approach, including both stationary and mobile intelligent agents, combined with the services likely to be offered by the emerging ubiquitous computing and intelligent home appliances, and applied to assist elderly people.
The traditional approach to care provision has been either to resort to support from relatives, or elderly care centers. However, these solutions have become increasingly inappropriate for the following reasons:

- Shifting the burden of responsibility onto relatives is increasingly impractical, given the fact that more and more family members have to work to secure steady incomes.
- Care centers are costly and invariably necessitate the relocation of the elderly people, often beyond their home communities. By so doing, elderly people may lose a degree of autonomy, and control over their daily lives.
- Many elderly people preserve enough robustness to be in their homes, a situation which is often preferable to the elderly people themselves, and as such, better for their welfare.

Therefore, the objective of the project is to leverage the potential of information society technologies, in particular, stationary and mobile intelligent agents and virtual organizations, to improve the quality of life, and care, for elderly people and their families. The intended platform will support the establishment of Virtual Elderly Assistance Communities.
Partners are "UNINOVA" (Center for Intelligent Robotics - Portugal), Faculty of Science (Amsterdam - Netherlands) , "Skill Consejeros de Gestión, S.L." - Spain, "Synkronix Incorporation Ltd." - United Kingdom, "Cámara Navarra de Comercio e Industria Unidad de Promoción y Desarrollo" - Spain, "Roundrose Associates Ltd." - United Kingdom.

2.2 Code Mobility

Code mobility can be defined as the capability to dynamically change the bindings between code fragments and the location where they are executed [44]. Code mobility is not a new concept. In the past, several mechanisms and facilities have been designed and

implemented to move code among the nodes of a network such as process migration and object migration. Examples are remote job submission [17] and the use of PostScript [55] to control printers or the UNIX "rsh" command.
Process migration mechanisms manage the bindings between the process and its execution environment to allow the process to resume its execution seamlessly in the remote environment. Process migration facilities were introduced at systems operational level to achieve load balancing across network nodes. Therefore, most of these facilities provide transparent process migration, i.e. the programmer has neither control nor visibility of migration.
Object migration makes it possible to move objects among address spaces, implementing a finer grained mobility with respect to systems providing migration at the process level. An example of system providing transparent migration is the COOL [64] object-oriented subsystem.
Process and object migration address the issues that arise when code and state are moved among the hosts of a loosely coupled, small scale distributed system. However, they are insufficient when applied in larger scale settings. Nevertheless, the two migration techniques have been taken as a starting point for the development of a new breed of systems, called mobile code systems, providing enhanced forms of code mobility.

2.2.1 Forms of Mobility

Existing mobile code systems offer two forms of mobility [44], characterized by the executing unit constituents that can be migrated:

Strong mobility [9] is the ability of mobile code system (called strong mobile code systems) to allow migration of both the code and the execution state of an executing unit to a different computational environment. The agent moves to another computer in the network and continues its execution from the point where it was before leaving the previous computer.

Weak mobility [9] is the ability of mobile code system (called weak mobile code systems) to allow code transfer across different computational environments; code may be accompanied by some initialization data, but no migration of execution state is involved. The agent moves to another computer in the network and starts execution from the beginning.

Strong mobility is supported by two mechanisms: *migration* and *remote cloning*. The migration mechanism suspends an executing unit, transmits it to the destination computational environment, and then resumes it. Migration can be either *proactive* or

reactive. In proactive migration, the time and destination for migration are determined autonomously by the migrating executing unit. In reactive migration, movement is triggered by a different executing unit that has some kind of relationship with the executing unit to be migrated, e.g., an executing unit acting as a manager of roaming executing units. The remote cloning mechanism creates a copy of an executing unit at a remote computational environment. Remote cloning differs from the migration mechanism in that the original executing unit is not detached from its current computational environment. As in migration, remote cloning can be either proactive or reactive.
Mechanisms supporting weak mobility provide the capability to transfer code across computational environments and either link it dynamically to a running executing unit or use it as the code segment for a new executing unit. Such mechanisms can be classified according to the direction of code transfer, the nature of the code being moved, the synchronization involved, and the time when code is actually executed at the destination site. As for direction of code transfer, an executing unit can either *fetch* the code to be dynamically linked and/or executed, or *ship* such code to another computational environment. The code can be migrated either as *stand-alone* code or as a *code fragment*. Stand-alone code is self-contained and will be used to instantiate a new executing unit on the destination site. Conversely, a code fragment must be linked in the context of already running code and eventually executed. Mechanisms supporting weak mobility can be either *synchronous* or *asynchronous*, depending on whether the executing unit requesting the transfer suspends or not until the code is executed. In asynchronous mechanisms, the actual execution of the code transferred may take place either in an *immediate* or *deferred* fashion. In the first case, the code is executed as soon as it is received, while in a deferred scheme execution is performed only when a given condition is satisfied e.g., upon first invocation of a portion of the code fragment or as a consequence of an application event.

2.2.2 Security Issues

Access and security problems appear from the instant an agent migrates from one site to another. It is precisely due to the property of migration, that mobile agents are exposed to different types of attacks [3]:

Unauthorized Access Malicious mobile agents can try to access the services and resources of the platform without adequate permissions. In order to thwart this attack, a mobile agent platform must have a security policy specifying the access rules applicable to various agents, and a mechanism to enforce the policy.

Masquerading A malicious agent assumes the identity of another agent in order to gain access to platform resources and services, or simply to cause mischief or even serious damage to the platform.

Denial of Service A malicious platform can cause harm to a visiting mobile agent by ignoring the agents request for services and resources that are available on the platform, by terminating the agent without notification, or by assigning continuous tasks to the agent so that it will never reach its goal.

Eavesdropping A malicious platform monitors the behavior of a mobile agent in order to extract sensitive information from it. This is typically used when the mobile agent code and data are encrypted.

Alteration A malicious platform tries to modify mobile agent information, by performing an insertion, deletion and/or alteration to the agents code, data, and execution state.

Prevention techniques for keeping the platform secure against a malicious mobile agent are discussed in detail in [3], [57], [56]:

Sandboxing In an execution environment, remote code, such as mobile agents and downloadable applets, is executed inside a restricted area called a "sandbox". A sandboxing mechanism enforces a fixed security policy for the execution of the remote code. The policy specifies the rules and restrictions that mobile agent code should comply with. The most common implementation of sandboxing is in the Java interpreter inside Java-enabled web browsers. A disadvantage of the sandboxing technique is that it increases the execution time of legitimate remote code.

Code Signing This technique ensures the integrity of the code downloaded from the Internet. Code Signing makes use of a digital signature and a one-way hash function. A well-known implementation of code signing is Microsoft Authenticode, which is typically used for signing code such as ActiveX controls and Java applets.

Proof-Carrying Code (PCC) In this technique, the code producer is required to provide a formal proof that the code complies with the security policy of the code consumer. PCC guarantees the safety of the incoming code providing that there is no flaw in the verification-condition generator, the logical axioms, the typing rules, and the proof-checker.

State Appraisal This technique ensures that an agent has not become malicious or modified as a result of its state alterations at an untrustworthy platform. The

author needs to anticipate possible harmful modifications to the agent's state and to counteract them within the appraisal function. Similarly, the sender, who sends the agent to act on his behalf, produces another state appraisal function that determines the set of permissions to be requested by the agent, depending on its current state and on the task to be completed. If both the author and the sender sign the agent, their appraisal functions will be protected against malicious modifications.

Path Histories The list of the platforms visited previously by the agent is the basis of trust that the execution platform has in the agent. Depending on the information in the Path History, the new platform can decide whether to run the agent and what privileges should be granted to the agent. The main problem with the Path History technique is that the cost of the path verification process increases with the path history.

Even if several security techniques already exist, none of these provides an optimal solution for all scenarios. Only a combination of various techniques may yield powerful solutions.

2.2.3 Design Paradigms

Design paradigms are described in terms of interaction patterns that define the relocation of and coordination among the components needed to perform a service. Let us consider a scenario where a computational component A, located at site S_A needs the results of a service. The existence of another site S_B, which will be involved in the accomplishment of the service, is assumed.
There are four main design paradigms exploiting code mobility: *client-server*, *remote evaluation*, *code on demand*, and *mobile agent*. These paradigms are characterized by the location of components before and after the execution of the service, by the computational component which is responsible for execution of code, and by the location where the computation of the service actually takes place. The presentation of the paradigms is based on a metaphor (introduced by G.Vigna et al in [44]) where two friends Laila and Clara interact and cooperate to make a lemon cake. In order to make the cake (the results of a service), a recipe is needed (the know-how about the service), as well as the ingredients (movable resources), an oven to bake the cake (a resource that can hardly be moved), and a person to mix the ingredients following the recipe (a computational component responsible for the execution of the code). To prepare the cake (to execute the service) all these elements must be collocated in the same home (site). In the follow-

Paradigm	Before		After	
	S_A	S_B	S_A	S_B
Client-Server	A	know-how, resource, B	A	know-how, resource, **B**
Remote-Evaluation	know-how, A	resource, B	A	*know-how*, resource, **B**
Code on Demand	resource, A	know-how, B	resource, *know-how*, **A**	B
Mobile Agent	know-how, A	resource	-	*know-how*, resource, **A**

Table 2.3: Mobile Code Paradigms

Table (see 2.3) shows the location of the components before and after the service execution. For each paradigm, the computational component in bold face is the one that executes the code. Components in italics are those that have been moved.

ing, Laila will play the role of component A, i.e., she is the initiator of the interaction and the one interested in its final results.

Client-Server (CS)

> *Laila would like to have a lemon cake, but she doesn't know the recipe, and she does not have at home either the required ingredients or an oven. Fortunately, she knows that her friend Clara knows how to make a lemon cake, and that she has a well supplied kitchen at her place. Since Clara is usually quite happy to prepare cakes on request, Laila phones her asking: "Can you make a lemon cake for me, please?" Clara makes the lemon cake and delivers it back to Laila.*

The client-server paradigm depicted in figure 2.1 is well-known and widely used. In this paradigm, a computational component B (the server) offering a set of services is placed at site S_B. Resources and know-how needed for service execution are hosted by site S_B as well. The client component A, located at S_A, requests the execution of a service with an interaction with the server component B. As a response, B performs the requested service by executing the corresponding know-how and accessing the involved resources collocated with B. In general, the service produces some sort of result that will be delivered back to the client with an additional interaction. In this paradigm mobility is not available.

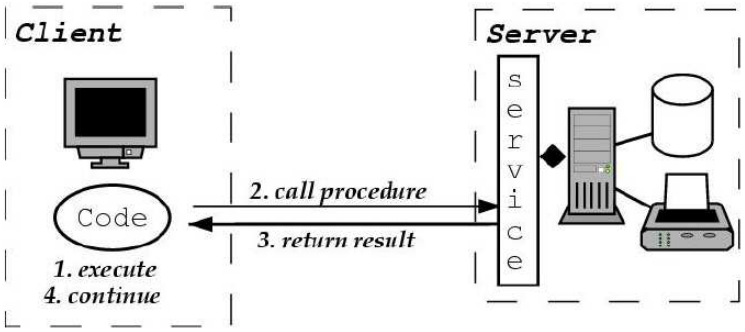


Figure 2.1: Client Server Paradigm

Remote Evaluation (REV)

> *Laila wants to prepare a lemon cake. She knows the recipe but she has at home neither the required ingredients nor an oven. Her friend Clara has both at her place, yet she doesn't know how to make a lemon cake. Laila knows that Clara is happy to try new recipes, therefore she phones Clara asking: Can you make a lemon cake for me? Here is the recipe: take three eggs.... Clara prepares the lemon cake following Lailas recipe and delivers it back to her.*

In the REV paradigm depicted in figure 2.2, a component A has the know-how necessary to perform the service but it lacks the resources required, which happen to be located at a remote site S_B. Consequently, A sends the service know-how to a computational component B located at the remote site. B, in turn, executes the code using the resources available there. An additional interaction delivers the results back to A. In this case we do not really have a code transfer but an execution transfer.

Code on Demand (CoD)

> *Laila wants to prepare a lemon cake. She has at home both the required ingredients and an oven, but she lacks the proper recipe. However, Laila knows that her friend Clara has the right recipe and she has already lent it to many friends. So, Laila phones Clara asking: Can you tell me your lemon cake recipe? Clara tells her the recipe and Laila prepares the lemon cake at home.*

In the COD paradigm dpicted in figure 2.3, component A is already able to access the resources it needs, which are co-located with it at S_A. However, no information

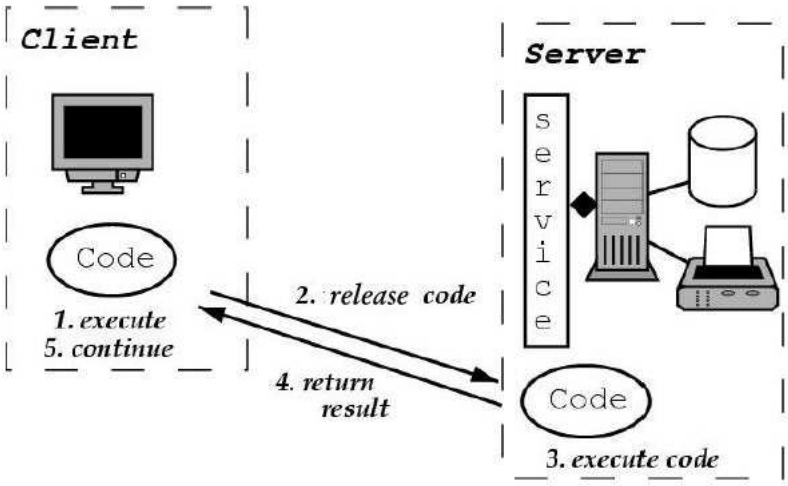


Figure 2.2: Remote Evaluation Paradigm

about how to manipulate such resources is available at S_A. Thus, A interacts with a component B at S_B by requesting the service know-how, which is located at S_B as well. A second interaction takes place when B delivers the know-how to A, that can subsequently execute it. In this case we have an execution transfer rather then a code transfer.

Mobile Agent (MA)

> *Laila wants to prepare a lemon cake. She has the right recipe and ingredients, but she does not have an oven at home. However, she knows that her friend Clara has an oven at her place, and that she is very happy to lend it. So, Laila prepares the lemon batter and then goes to Claras home, where she bakes the cake.*

In the MA paradigm depicted in figure 2.4, the service know-how is owned by A, which is initially hosted by S_A, but some of the required resources are located on S_B. Hence, A migrates to S_B carrying the know-how and possibly some intermediate results. After it has moved to S_B, A completes the service using the resources available there. The mobile agent paradigm is different from other mobile code paradigms since the associated interactions involve the mobility of an existing computational component. In other words, while in REV and COD the focus is on the transfer of the code execution between components, in the mobile agent paradigm a whole computational component is moved to a remote site, along with its state, the code it needs, and some resources required to

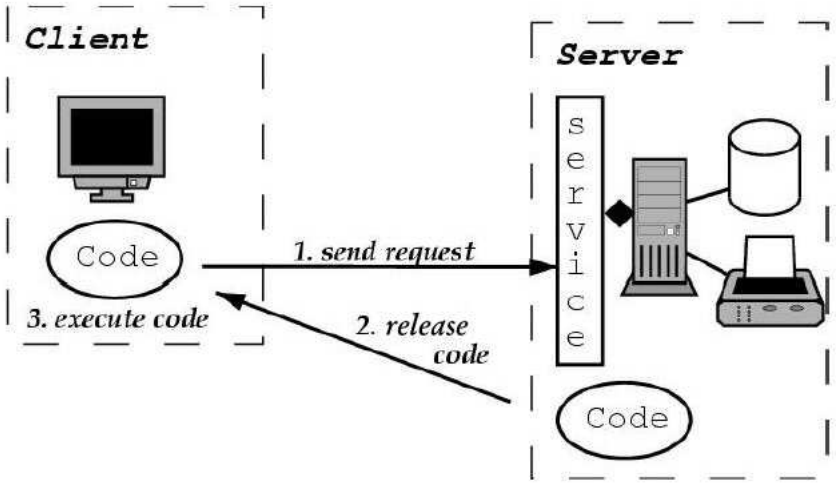


Figure 2.3: Code on Demand Paradigm

perform the task.

2.3 Mobile Agents Patterns

This section deals with the aforementioned described mobile code paradigm, the mobile agents. [2] proposes a model of an agent system, in which the agents possess a set of characteristic abilities:

Creation This involves the ability of certain agents in creating other agents, to perform some specified tasks. The resulting agents can be created to run locally or remotely.

Execution Agents need to be able to execute in order to carry out their tasks. Execution is usually done through an interpreter that supports a runtime environment within which an agent can function. This environment is usually specified when the agent starts up in the machine.

Resource Access This should implement ways in which an agent accesses certain local resources as well as resources being carried with the agent. Hence, there are two types of resources present within the execution environment, resulting in two very different security concerns. The types of resources being considered could include physical resources as well as logical resources and controlling access to those resources can be done using many different mechanisms including access

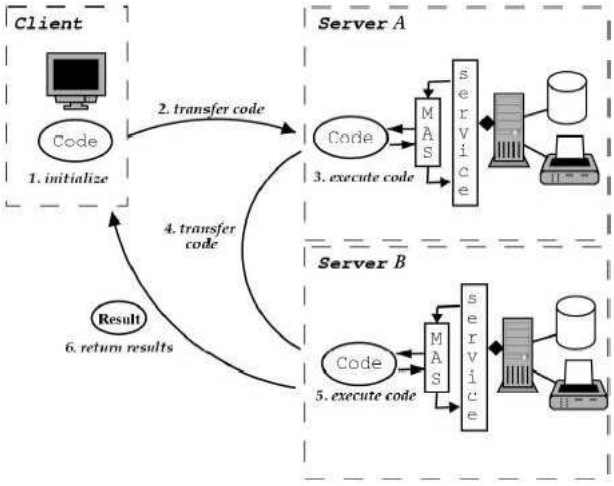


Figure 2.4: Mobile Agents Paradigm

control or capabilities. An example would be restricting access to the CPU and other resources through checking during the interpretation/execution phase.

Migration This implements how an agent can move around from one machine to the next on its own initiative, while carrying out its task. This aspect of the architecture is what makes agents very interesting and what gives them unique properties. This migration process can be affected by the move being unrestricted (i.e. from any machine to another) or restricted (i.e. from the home machine to the server machine and back). It also involves how and in what form the data is being carried around within the agent. This point refers to migrating agent resources (in addition to code migration). There is also the issue of implicit versus explicit transfer of an agent's state.

Communication This takes care of the fact that agents need to interact with other agents to provide their services. There are two types of communication that could be available. One type would be local while the other is remote. Communication may also serve as a way to transfer objects (and agents) between other agents. This also includes the issue of creating synchronization primitives between agents that can help in organizing their efforts when obtaining a certain service in a cooperative manner.

Language Support This involves the issue of interpreted versus compiled languages. It also involves support for just one specialized language versus the support of many different languages to be used in programming agents.

Additional Services Services like authentication, name service, check-pointing, as well as other system built-ins. The services depend on the communication and on the access treatment (a distant one if necessary). The issue here is that some of these services can be implemented through the system while others are easier to implement using agents.

2.3.1 Agents Platform and Standardization

Nowadays we can find a big number of different agent systems. Table 2.4 at the end of this section shows only a small overview. Usually agent systems make available basic classes for the user, with which he is able to build his own agents. These agents can then be executed on the provided agent platforms. A large problem of today's agent systems is the lack of standardization. An agent, who was implemented for an agent system, does not let itself be executed on another agent system. For this reason, standards for agent systems were defined. In the following we will present the FIPA standard as well as the MASIF standard.
An agent framework is a software environment in which software agents run. It provides support for software agents to execute, to manage their execution, to access system resources, and to guarantee integrity and protection of both agents and the platform itself. Agents platforms also provide support for migration, naming, location and communication services. Examples of agent frameworks are: "IBM's Aglets", "ObjectSpace's Voyager", "IKV's Grasshopper" etc.
These systems differ widely in architecture and implementation, thereby impeding interoperability and rapid deployment of mobile agent technology in the marketplace. To promote interoperability, some aspects of mobile agent technology needs to be standardized.
Currently there are two standards for mobile agents technology: The Object Management Group's Mobile Agent System Interoperability Facility (MASIF) and the specifications promulgated by the Foundation for Intelligent Physical Agents (FIPA). Object Management Group (OMG) was formed in 1989 by 11 companies including 3Com, HP, Canon, Sun, Unisys and American Airlines. Now it includes about 800 members. It is a no-profit corporation. Its most famous standard is certainly CORBA. The OMG was formed to create a component-based software marketplace by hastening the introduction of standardized object software. Its charter includes the establishment of industry guidelines and detailed object management specifications to provide a common framework for application development. Conformance to these specifications will make it possible to develop a heterogenous computing environment across all major hardware

platforms and operating systems.
In 1995 the OMG started working on a standard, called Mobile Agent Facility (MAF), in order to promote interoperability among agent platforms. In 1997, a joint submission by IBM, General Magic, The Open Group, GMD FOKUS, and etc. was presented to the OMG. And the standard's name was changed from MAF to Mobile Agent System Interoperability Facility (MASIF) [28]. In 1998 this specification was accepted as an OMG standard. The current edition was issued in 2000. MASIF defines two interfaces:

- MAFAgentSystem
- MAFFinder

Both interfaces are specified within the *Interface Definition Language (IDL)*. Basic methods are defined within the *MAFAgentSystem*, which every agent system should make available (see Source Code 1).

Source Code 1 — *Interface MAFAgentSystem*

```
Interface MAFAgentSystem {
void create_agent(...); // creates an agent
void receive_agent(...); //receives an agent
void suspend_agent(...); //suspends the agent execution
void resume_agent(...); //resumes the agent execution
void terminate_agent(...); //deletes an agent

void terminate_agent_system(...); //closes the agent system

AgentStatus get_agent_status(...); //asks the agent status

NameList list_all_agents(); // returns the list of all the agents
NameList list_all_places(); // returns contexts within the system
};
```

The *MAFFinder* makes available methods in order to find agents or rather agent systems (see Source Code 2).

Source Code 2 — *Interface MAFFinder*

```
Interface MAFFinder {
//registering methods
void register_agent(...);
void register_agent_system(...);
void register_place(...);

//searching methods
Locations lookup_agent(...);
Locations lookup_agent_system(...);
Locations lookup_place(...);

//unregistering methods
void unregister_agent(...);
void unregister_agent_system(...);
void unregister_place(...);
};
```

IKV's Grasshopper agent system (version 1 available in 1998) is conforming to MASIF. It is a mobile agent and runtime platform developed in Java, built upon a distributed object-oriented middleware. As submitters of the MASIF standard, IBM and GMD FOKUS agreed to implement the MASIF specification within their own agent platforms. (GMD stands for German National Research Center for Information Technology, FOKUS stands for Research Institute for Open Communication Systems). There is another mobile agent system called Secure and Open Mobile Agent (SOMA) System, developed by Universita' di Bologna in Italy. SOMA has been developed, "closely considering compliance with MASIF."
The Foundation for Intelligent Physical Agents (FIPA) [41] was formed in 1996 to produce software standards for heterogenous and interacting agents and agent-based systems. It is a non-profit association formed under Swiss law. Its members include companies and universities. FIPA identified a list of agent technologies deemed to be specifiable in 1997 and standardization work started. There is a set of specifications called FIPA 97 and another called FIPA 98, both are now on Obsolete Status. The current specification is FIPA 2000, half of which is in the Preliminary Status, another half on Experimental Status. The focus of FIPA is on Intelligent Agents, Agent Coop-

eration and the whole environment of that software area. This is in contrast to the work of the MASIF specification, which is mostly applied to Mobile Agents, although these two main standardization organizations have common points and therefore there exists a unification of their concepts [47].

	Aglets	**Concordia**	**Grasshopper**	**J-Seal**	**Zeus**
Version	2.0.2	1.1.7	2.2.4	2	1.9.4
Producer	IBM	Mitsubishi Electric	IKV++	CoCo Software	British Telecommunication
Particularity	MASIF,FIPA compatible		Graphic development environment		
Open Source	x				x
Mobility	week	strong	week	week	week
Communication	synchronous, asynchronous	asynchronous, collaboration	RMI, IIOP, CORBA, MAF IIOP	asynchronous	asynchronous TCP/IP, FIPA ACL
Security Concept	SSL, X.509	SSL	SSL, X.509	SSL	none

Table 2.4: Overview: Agent Systems

2.4 Conclusion and Personal Reflections

"A part of that force which constantly intends evil and yet creates good."
(Faust I, line 1335)
The Curse and Blessing of Innovation / Science Within the Constraints of the Market

Let us consider a dilemma, one that requires a decision, because it cannot be avoided. At the dawn of mobile agents as a technology, their role as a catalyst of innovation is regarded with skepticism, if not outright denial. It is said that this technology is a bit expensive and is not able to achieve what the orthodox has or will. Additionally, and most condemning, this technology is not free from abuse and, in fact, may clear the path for even greater abuse. Neither the arguments nor the underlying discussion are new. The matter at hand is the "eternally new" promised according to the sense - or nonsense - of science and innovation. Nothing more and nothing less is involved in this debate; a debate on ethics and science measured on criteria that are out of its

scope. Because science in and among itself does not have a norm with which to aid in its orientation. The objectives of science are not pursued through individual curiosity. Rather more crucial are the priorities set by those people and institutions who invest in science financially. Computer science and, specifically, application development is no exception; profitability is the key objective. On the one hand, science carries a burden of responsibility for mankind. On the other hand, research does not exist in a vacuum. It is called upon to anticipate and handle the consequences of the results of an application. Yet it does not offer a recipe to control the errors and corruptibility of man. Can one accept the Evil in the creation of the Good? Granted, this is an academic question. However, the risk that this young child called "Mobile Agent" falls down this well, has been with us for some time. The question is then "resignation or action"?

A layman understands in an instant what mobile agents are, when he is given the technical definition that what is being referred to are autonomous applications that move from computer to computer to make use of their local resources. He correctly assumes then that they are benevolent computer viruses. The term itself, "agent", suggests secrecy, anonymity, espionage, illegal activity. Equally suspicious is the association with virus, be it benign or otherwise. The fact is that this technology, through its very terminology unfortunately conveys that, if it is not a problem in itself, it, at least, has one. All of the positive promises of innovation are moot when faced with the fact that this technology already exists as a negative occurrence. Computer viruses cannot be dismissed and have demonstrated that their application cannot be always controlled. That this heretofore sinister entity should be the potential bearer of good is a concept that will take some getting used to. Clearly, compared with computer viruses, mobile agents represent the more significant technological challenge. The "functional" abuse should be motivation to tackle the intended progress and convince the skeptics, who fear that the New endangers the Established. The desire to implement technology simply because it is more innovative, it expands horizons, it could serve as a catalyst, is considered by many to be unreasonable as no one wishes to find himself in unfamiliar territory when change is not called for. Crises are thus typically the triggers for technological evolution. The demise of "antiquated" technology is the birth of a "new" one. Crisis and innovation appear so inseparable that "makers" of innovation are considered cynics. When the New asserts itself at the cost of the Old, the results are the same. There are as many examples of this dramatic relationship between tradition and innovation as there are technologies in the world. Some of which are named here in order to somewhat diffuse the dogmatic seriousness of this debate. When the telephone, for instance, was introduced, it was said to be of little use as a means of communication. The market for computers at the dawn of the 1940's amounted to a measly five machines and, at the end of the 70's, it was

unimaginable that a computer would ever be a desirable fixture in a private household. The blessings of the Internet far outweighed its curse. And we will call upon the services of "mobile agents" as readily in the future as we use a remote control to operate a television today. Until that day, the problems of security remain to be resolved. It is futile to continuously stress that "mobile agents" be only allowed to roam in approved areas and that they should only be enabled to access forthcoming data, as long as this assertion merely conveys a noble notion and not a realization. The most difficult challenge is not the development of an effective control mechanism to identify the mobile agent and make their tasks transparent and dependent on the authorization of their destination. It is, instead, the minimizing of risks through abuse and technical or human error. An absolute guaranty of security will never be realistic (can man ever truly protect itself from man?) A dissertation, such as this one, should not be evaluated on the feasibility of market introduction for the technology it is dedicated to. The young must be allowed to break new ground in the carefree presumptuousness that is otherwise out of reach through external constraints. When one only attends to problems that have a certain solution, then one falls short of his potential.

The ambition of this thesis is to bring a solution to certain problems we have shown above in this chapter. In the section "Reasons for Downfall" of mobile agents one disadvantage was that mobile agents are difficult to design and another one that they are difficult to test. One of the solutions we propose, offers a specification language, namely based on the higher order π-calculus, associated with a code generator, to facilitate the design of mobile agents and also the verification and validation of such systems.

Chapter 3

Calculi and Specification Languages for Mobile Agents

One of the system modeling issues is its verification and validation (V&V). There are three major classes for the verification of concurrent systems: static, dynamic, and formal analysis.

Static analysis techniques are those which directly analyze the form and structure of a product without executing the product. Reviews, inspections, audits and data flow analysis are examples of static analysis techniques. Static analysis techniques are traditionally applied to software requirements, software design and source code. They may also be applied to test documentation, especially test cases, to verify their traceability to the software requirements, their adequacy to fulfill test requirements, and their accuracy.

Dynamic analysis techniques involve execution, or simulation, of a development activity product to detect errors by analyzing the response of a product to sets of input data. For these techniques, the output values, or ranges of values, must be known. Testing is the most frequent dynamic analysis technique. Prototyping, especially during the software requirements V&V activity, can be considered a dynamic analysis technique; in this case the exact output is not always known but enough knowledge exists to determine if the system response to the input stimuli meets system requirements.

Formal analysis is the use of rigorous mathematical techniques to analyze the algo-

rithms of a solution. Sometimes the software requirements may be written in a formal specification language (e.g., π-calculus, Z, CSS, CSP, Petri nets or some kinds of automata) which can be verified using a formal analysis technique like proof-of-correctness. The term formal often is used to mean a formalized process, that is, a process that is planned, managed, documented, and is repeatable. In this sense, all software Verification and Validation techniques are formal, but do not necessarily meet the definition of the mathematical techniques involving special notations and languages.

The most difficult part in a V&V process of a concurrent model is the possibly state-space explosion of the system, that makes the verification impossible for a human being. Therefore the verification has to be automated, that will impose the use of a formal model which can be manipulated automatically.
Talking about mobile systems, there exists several formalisms that allow one to model and reason about mobile systems. One approach is to model mobility as the changing of communication paths; another is to use explicit notions of location and migration of processes between these locations.
This chapter will present three formal languages that can express mobility: the π-calculus with some of its extensions, the ambient calculus and the mobile and dynamic Petri Nets. The interest is to show how each formalism can be assigned to different design paradigms introduced in 2.2.3. The criteria, upon which the selection is based, are:

- the syntax i.e., in our case, the power to express exchanges with synchronization,
- the semantics i.e. the power of validation; we are interested in an operational semantics,
- the scope of the terms.

The basic π-calculus expresses mobility as the changing of communication paths, whereas the ambient calculus, has explicit location concepts.
All the formal languages build on a minimal set of primitives; they focus on the communication between processes and abstract away other computation issues. Formal semantics and equivalence relations have been defined for each language, allowing one to reason about systems. The equivalence relations are defined in terms of the observable behavior of a system, which in most cases means the possible communication of the system with the environment. This provides a formal framework for modeling, reasoning about and verification of systems and programs.

A full formal description of the languages will not be given; only their syntax and the interesting part of their semantics will be presented. The purpose of this presentation is to show how the two formalisms work, what they can do w.r.t. one another and why we chose the higher-order π-calculus in order to express our specifications of the mobile agents. For a full formal description of the languages the reader is referred to the literature.

3.1 The π-Calculus and its Extensions

In this work we will adopt the notations and the definitions introduced by Milner in [68, 67], and given by Parrow in [80, 70]. The π-calculus is a mathematical model of processes whose interconnections change as they interact. The π-calculus aims at "the challenge of defining an underlying model, with a small number of basic concepts, in terms of which interactional behavior can be rigorously described" (Milner p. 3, Introduction in [68])". The basic computational step is the transfer of a communication link between two processes; the recipient can then use the link for further interaction with other parties. This makes the calculus suitable for modeling systems where the accessible resources vary over time.
The definition of the π-calculus is not steady and many different evolution of π can be found in the literature. The π-calculus is actually more a family of calculi than just a unique calculus. Such evolutions, briefly summarized in [91], include asynchronous, internal or receptive version of the π-calculus; extensions with primitive for testing the (in)equality of names; etc. In addition, several process calculi based on name-passing has been proposed: the fusion calculus of Parrow and Walker [81], a simplification of π with a more symmetric form of communication; the spi-calculus of Abadi and Gordon [1], an extension of π designed for the description and analysis of cryptographic protocols, the join-calculus of Fournet, Gonthier et al [43]; the blue-calculus of Boudol [18]; etc.
There are several extensions of the π-calculus. Three of them are: the *monadic* π-calculus, which is the basic form and allows only a single channel name to be sent in a message. The *polyadic* π-calculus extends the monadic to allow a tuple of channel names to be sent in a single message. The *higher-order* π-calculus also allows agents to be sent in a message, and it can thus model mobility more directly.

3.1.1 Basic Definitions - The Monadic π-Calculus

Let us consider:

- an infinite set of *names* $\mathcal{N}$, ranged over by a, b, ..., z, which will function as all of communication gates, variables and data values,
- a set of *identifiers* (we will call them also *agents* or *processes*) ranged over by A, each with a fixed nonnegative arity.

The *agents*, ranged over by P, Q,... are defined in Table 3.1.1. According to this table we distinguish different agent behaviors:

1. The *Output Prefix* $\bar{a}x.P$ can send the name x via the name a and continue as P.
2. The *Input Prefix* $a(x).P$ can receive any name via a and continue as P with the received name substituted for x. For instance, $a(x).\bar{b}x.0$ can receive any name via a, send the name received via b, and become inactive, while $a(x).\bar{x}b.0$ can receive any name via a, send b via the name received, and become inactive.
3. The *Silent Prefix* $\tau.P$ represents an agent that can evolve to P without interaction with the environment. Parrow in [80] uses α, β to range over $a(x)$, $\bar{a}x$ and τ and call them *Prefixes*.
4. The empty agent 0 cannot perform any action.
5. The *Sum* $P + Q$ represents an agent that can enact either P or Q. For instance, $a(x).\bar{x}y.0 + \bar{b}z.0$ has two capabilities: to receive a name via a, and to send z via b. If the first capability is exercised and u is the name received via a, then the continuation is $\bar{u}y.0$; the capability to send z via b is lost. If, on the other hand, the second capability is exercised, then he continuation is 0, and the capability to receive via a is lost.
6. The *Parallel Composition* $P|Q$ represents the combined behavior of P and Q executed in parallel, where P and Q can proceed independently and interact via shared names. For instance, $(a(x).\bar{x}y.0 + \bar{b}z)|\bar{a}u.0$ has four capabilities: to receive a name via a, to send z via b, to send u via a, and to evolve invisibly as an effect of an interaction between its components via the shared name a.
7. The *Match* $[x = y]P$ can evolve as P if x and y are the same name, and can do nothing otherwise. For instance, $a(x).[x = y]\bar{x}z.0$, on receiving a name via a, can send z via that name just if that name is y; if it is not, the process can do nothing further, i.e. it is blocked.
8. The *Restriction* $(\nu x)P$ behaves as P but the scope of the name x is restricted to P. Components of P can use x to interact with one another but not with other

processes. For instance, $(\nu x)((a(x).\bar{x}y.0 + \bar{b}z.0)|\bar{a}u.0)$ has only two capabilities: to send z via b , and to evolve invisibly as an effect of an interaction between its components via a. The scope of a restriction may change as a result of interaction between agents.

9. The *Replication* $!P$ can be thought of as an infinite composition $P|P|...$ or, equivalently, an agent satisfying the equation $!P = P|!P = !P|P$. replication is the operator that makes it possible to express infinite behaviors. For example, $!a(x).!\bar{b}y.0$ can receive names via a repeatedly, and can repeatedly send via b any name it does receive.

10. The *Identifier* $A(y_1, ..., y_n)$, where n is the arity of A, has a *Definition* $A(x_1, ...x_n) \stackrel{\text{def}}{=} P$ where the x_i must be pairwise distinct, and the intuition is that $A(y_1, ..., y_n$ behaves as P with y_i replacing x_i for each i. A Definition can be thought of as an agent declaration, $x_1, ...x_n$ as formal parameters, and the Identifier $A(y_1, ..., y_n)$ as an invocation with actual parameters $y_1, ..., y_n$.

Prefixes	α	::=	$\bar{a}x$	Output
			$a(x)$	Input
			τ	Silent
Agents	P	::=	0	Nil
			$\alpha.P$	Prefix
			$P + P$	Sum
			$P\|P$	Parallel
			$[x = y]P$	Match
			$(\nu x)P$	Restriction
			$!P$	Replication
			$A(y_1, ..., y_n)$	Identifier
Definitions			$A(x_1, ..., x_n)$	(where $i \neq j \Rightarrow x_i \neq x_j$)

Table 3.1.1: The syntax of the π-calculus

Definition 3.1 (Binding) *[90] There are three binding operators:*

- *the* ***input prefix*** $a(x)$ *(which binds* x*)*
- ***restriction*** $(\nu x)P$
- ***identifier*** $A(y_1, ...y_n)$,

The occurrence of x *is* binding *with* scope P*. An occurrence of a name in an agent is* bound *if it is, or it lies within the scope of, a binding occurrence of the name.*
An occurrence of a name in an agent is free *if it is not bound.*
Let us define the free names $fn(P)$*, and the* bound names $bn(P)$ *of an agent. Extended to prefixes; we note:*

$$\begin{aligned} bn(a(x)) &= \{x\},\ fn(a(x)) = \{a\} \\ bn(\bar{a}x) &= \emptyset,\ fn(\bar{a}x) = \{a, x\} \end{aligned}$$

For instance,

$$fn((\bar{a}x.0 + \bar{b}y.0)|\bar{c}u.0) = \{a, x, b, y, c, u\}$$

and

$$fn((\nu a)((a(x).\bar{x}y.0 + \bar{b}u.0)|(\nu v)(\bar{a}v.0))) = \{y, b, u\}$$

Definition 3.2 (Substitution) *[90] A* substitution *is a function from names to names. One writes* x/y *for the substitution that maps* y *to* x *and its identity for all other names, and in general* $\{x_1...x_n/y_1...y_n\}$*, where the* y_i *are pairwise distinct, for a function that maps each* y_i *to* x_i*.*

For instance, let us consider
$\bar{u}a|u(x).P(x)$ with $P(y) = \bar{u}y.0$ After the communication and substitution the process P becomes $P(a) = \bar{a}y.0$.

Operational Semantics

The operational semantics of the π-calculus is given through a labelled transition system, where transitions are of kind $P \xrightarrow{\alpha} Q$ for some set of actions ranged over by α.
There are three labels for the transitions: the silent step τ, the input action $a(x)$ and the output action $\bar{a}x$.

Output action: $\bar{a}x.P \xrightarrow{\bar{a}x} P$ means that after having sent message x over channel a, process $\bar{a}x.P$ behaves like P.
Input Action: $a(x).P \xrightarrow{a(u)} P\{u/x\}$ means that if name u is sent over channel a, then the process $a(x).P$, waiting for a process or channel name on a, receives it and replaces the process or channel name by u, it then behaves like P, where all occurrences of x are replaced by u.
The communication rule is the most conspicuous for a mobility system. The interaction between two processes is given by the "Communication rules Com1 and Com2" in [90]:
$Com1$:

$$\frac{P \xrightarrow{\bar{a}x} P', Q \xrightarrow{a(y)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{x/y\}}$$

$Com2$:

$$\frac{P \xrightarrow{a(y)} P', Q \xrightarrow{\bar{a}x} Q'}{P|Q \xrightarrow{\tau} P'\{x/y\}|Q'}$$

i.e. an output action causes P to become P', the corresponding input action causes Q to become Q' then P and Q in parallel become P' and Q' in parallel.
The labeled transition semantics is given below:

$$\texttt{STRUCT} \quad \frac{P' \equiv P, P \xrightarrow{\alpha} Q, Q \equiv Q'}{P' \xrightarrow{\alpha} Q'} \tag{3.1}$$

$$\texttt{PREFIX} \quad \frac{}{\alpha.P \xrightarrow{\alpha} P} \tag{3.2}$$

$$\texttt{SUM} \quad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \tag{3.3}$$

$$\texttt{MATCH} \quad \frac{P \xrightarrow{\alpha} P'}{if\ x = x\ then\ P \xrightarrow{\alpha} P'} \tag{3.4}$$

$$\texttt{PAR} \quad \frac{P \xrightarrow{\alpha} P', bn(\alpha) \cap fn(Q) = 0}{P|Q \xrightarrow{\alpha} P'|Q'} \tag{3.5}$$

$$\texttt{COM} \quad \frac{P \xrightarrow{a(y)} P', Q \xrightarrow{\bar{a}x} Q'}{P|Q \xrightarrow{\tau} P'\{x/y\}|Q'} \tag{3.6}$$

$$\texttt{RES} \quad \frac{P \xrightarrow{\alpha} P', x \notin \alpha}{(\nu x)P \xrightarrow{\alpha} (\nu x)P'} \tag{3.7}$$

For a detailed explanation of the operational semantics rules see [80].
The particularity of the π-calculus is to allow names of channels to be passed as parameters. If a process moves, its neighborhood changes and with it the channels it uses for communication. Transition $a(x).P \stackrel{a(y)}{\rightarrow} P\{y/x\}$ means that message y is sent along the channel a. If we consider that y is not a simple value but a channel name, then the resulting process $P\{y/x\}$ is able to use this name as a channel for further communications. The actual value of the channel is instantiated during the execution of the process.

3.1.2 Formal Method for Mobility - The Higher Order π-Calculus

This section starts by reviewing the definition of the higher-order π-calculus (HOπ); an extension of the π-calculus from 3.1.1. The syntax and the operational semantics of this calculus will be given along with a sequence of examples that illustrates mobility in the higher-order π-calculus.
The π-calculus provides a conceptual framework for understanding mobility, and mathematical tools for expressing mobile systems and reasoning about their behavior. The first questions we should ask are: What is mobility, what are the entities that move, and in what space do they move? Two kinds of mobility can be distinguished: "name mobility", which corresponds to the Remote Evaluation paradigm (see 2.2.3) and "action mobility", which corresponds to the Mobile Agent paradigm (see 2.2.3). In the first instance, it refers to *links* that move in an abstract space of *linked processes*. For example, in the World Wide Web, the hypertext links can be created, can be passed and can disappear just like the connections between mobile phones. In the second kind of mobility, it is an *agent* or a *process* that moves in an abstract space of linked processes. For instance, a piece of code that can be an agent, can move from a machine to another over the network in order to accomplish a task; mobile devices can acquire new functionalities using, for example, the Jini technology.
The π-calculus addresses the first kind of mobility: it directly expresses movement of links in a space of linked processes. There are two kinds of basic entities in the π-calculus: names and processes. Names are names of links, and processes can interact by using names they share.
The higher-order π-calculus (HOπ) treats the second kind of mobility, where it is the processes (agents) that move. The π-Calculus and its extensions are process algebras that focus on process mobility. Processes communicate using channels, which define the configuration of the system. Processes send a channel name in the monadic π-Calculus, tuples of channel names in the polyadic π-Calculus, and tuples of processes and channel

names in the higher-order π-Calculus (HOπ) [90].
Sangiorgi identifies a first-order paradigm and a higher-order paradigm for mobility in process algebra. The notion of mobility in process algebra is achieved by sending messages that change the communication interface between components of the system. The first-order paradigm allows gates or names to be transmitted as messages. After the transmission of a gate, the communication can take place through this gate. The higher-order paradigm allows processes (parameterized or not) to be passed as values in a communication. After a process has been transmitted, it can start its execution. This is a process-passing mechanism. Sangiorgi [90] proves that the expressiveness of higher-order and first-order π-Calculus are the same.

Syntax

A higher-order π-Calculus process is given by the following *syntax*:

$$\boxed{\begin{array}{l} P ::= \sum_{i\in(I)} \alpha_i.P_i \mid P_1|P_2 \mid P_1 + P_2 \mid \nu\, x.P \mid [x = y]P \\ \alpha ::= x(U) \mid \overline{x}K \end{array}}$$

Here I is a finite indexing set; in the case $I = \emptyset$ we write the sum as 0. K and (U) stand for any tuple of agent (process) or (channel) name and:

- $\bar{x}K.P$ can send the name or process K via the name x and continue as P.
- $x(U).P$ can receive any name or agent U and continue as P with the received name substituted for U.
- in the composition $P_1|P_2$, the two components can proceed independently and interact via shared names or processes.
- $\nu\, x.P$ is called the restriction and means that the scope of name x is restricted to P.
- in the sum $P_1 + P_2$ either P_1 or P_2 can interact with other processes.
- The matching $[x = y]P$ denotes the activation of a process which is selected by other processes depending on a condition ($[x = y]$) This operator $=$ is a boolean predicate defined in this algebra.

The difference between first-order and higher-order π-Calculus resides in the fact that parameters can be channels and/or processes in the higher-order π-Calculus, while in

first-order π-Calculus only channels can be passed as parameters. That means every channel name in a first-order calculus can be replaced by a parameterized process (if necessary) in higher-order calculus.
Example: in $\overline{x}P.Q|x(X).X$, once the interaction between the two processes has taken place, the resulting process is $Q|P$. Indeed, process $x(X).X$ was waiting for X to be sent along channel x, i.e., it was waiting for a process X defining its subsequent behavior.

Arity and Substitution Issues

The question is, how to treat agents such as $\bar{x}K.P \mid x(U).Q$ where the arity of the output is not the same as the arity of the input. J. Parrow proposes in his work about the polyadic π-calculus [80] to adopt the notion of "sorting". His idea is that each name is assigned a *sort*, containing information about the agents that can be passed along that name.
In the polyadic π-calculus, sorts are also essential to avoid disagreement in the arities of tuples carried by a given name. We write $x : s$ to mean that x belongs to the subject sort s; this notation is extended to tuples along their components.
We apply his idea to the higher-order π-calculus in order to resolve the substitution issues generated by the fact that an agent can also be an object, a function, or a name, and that two or several agents can have a different arities.

Definition 3.3 *If $\mathcal{S}$ is a set of sorts, a sort context $\triangle$ is a function from $\mathcal{N}$ to $\mathcal{S}$. Then $\triangle(K)$ is a sort of K.*

In a simple system, a sort would be nothing more than a natural number, $\mathcal{S} = \mathcal{N}$, such that $\triangle(x)$ denotes the arity of x, i.e., the number of objects in any prefix where x is the subject. Formally the notation $\triangle \vdash P$ means that P conforms to $\triangle$, and the rules for inferring $\triangle \vdash P$ can be given by induction over the structure of P:

$$\frac{\triangle \vdash P\ ,\ \triangle(x) = n}{\triangle \vdash \bar{x}K.P}$$

$$\frac{\triangle \vdash P\ ,\ \triangle \vdash Q}{\triangle \vdash P|Q}$$

Using this idea the agent $\bar{x}K.P \mid x(U).Q$ is malformed when $\triangle(x)$ in one component is bigger or smaller then $\triangle(x)$ in the other component.

However, this simple scheme works for this particular example. To be able to capture not only the immediately obvious arity conflicts, but also any such conflicts that can arise during execution, more information must be added to the sorts. For each name, the number of objects passed along the name is not enough; the sort of each such object must also be included. In the example:

$$x(u).u(z) \mid \bar{x}y.\bar{y}K$$

the left component requires the sort of x to be one object (corresponding to u) which has the sort 1 because of the subterm $u(z)$. The right component requires the sort of x to be one object (corresponding to y) of sort of the agent K due to the subterm $\bar{y}K$. If the sort is 1, then the term above is well-formed, yet if the sort is different from 1, the term is malformed.
Arity conflicts can be arbitrarily deep, meaning that the sort of x must contain information about the agents which are passed along to their agents, which, in turn, are then passed along to their agents, etc... To resolve this problem we can associate with each sort S in $\mathcal{S}$ a fixed *agent sort* $ag(S)$ in $\mathcal{S}^*$ i.e., the agent sort is a (possibly empty) sequence of sorts. The intention is that if x has the sort S where $ag(S) = \langle S_1 ... S_n \rangle$, and $x(U_1 ... U_n)$ is a prefix, then each U_i has the sort S_i. With a slight abuse of notation the sorting rule for the input becomes:

$$\frac{\triangle \cup \{K \longrightarrow ag(\triangle(x))\} \vdash P}{\triangle \vdash x(K).P}$$

It should be read as follow: In order to establish that $x(K).P$ conforms to $\triangle$, find the agent sort S of x according to $\triangle$, and verify that P conforms to $\triangle$ where agent K is assigned sort S. The rule for output is:

$$\frac{ag(\triangle(x)) = \triangle(K), \triangle \vdash P}{\triangle \vdash \bar{x}K.P}$$

In order to establish that $\bar{x}K.P$ conforms to $\triangle$ it is enough to show that it assigns K the agent sort of x, and that P conforms to $\triangle$.

Definition 3.4 *Let us consider an* agent variable*, ranged over by Y, and let us extend the definition of agents to include the agent variables and the higher-order prefix forms.*

The notion of replacing an agent by an agent, $P\{Q/Y\}$ and the higher-order interaction rule gives that:

$$x(Y).P \mid \bar{x}Q.R \xrightarrow{\tau} P\{Q/Y\} \mid R$$

The substitution $P\{Q/Y\}$ is defined with alpha-conversion such that free names in Q do not become bound in $P\{Q/Y\}$.
In the following example:

$$x(P).y(u).P \mid \bar{x}\langle u(z).0\rangle.Q$$

the name u in the right hand-side component is not the same as the bound name u in the left hand-side component. The agent is alpha-equivalent to:

$$x(P).y(t).P \mid \bar{x}\langle u(z).0\rangle.Q$$

where the names are dissociated. Therefore a transition to $y(u).u(z) \mid Q$ is not possible. The binding in this example, corresponds to a *static* binding: the scope of a name is determined by its location within the agent.
The alternative *dynamic* binding, where the scope is determined only when the name is actually used, cannot use alpha-conversion and it is semantically very complicated.
An expressive power to the higher-order π-calculus brings the fact that a transmitted agent Q is duplicated by the interaction. Consider the example:

$$x(Y).(Y|Y) \mid \bar{x}Q.P \xrightarrow{\tau} Q|Q|P$$

Replication and *Recursion* are derivable constructs. Consider:

$$D = x(Y).(Y \mid \bar{x}Y)$$

D accepts an agent along x, and will start that agent and also retransmit it along x. Let P be any agent with $x \notin fn(P)$, and

$$R_P = (\nu x)(D \mid \bar{x}\langle P|D\rangle)$$

Then R_P behaves as $!P$ since it will reproduce an arbitrary number of P:

$$R_P \xrightarrow{\tau} (\nu x)((P|D) \mid \bar{x}\langle P|D\rangle) \;\equiv\; P \mid R_P \xrightarrow{\tau} P \mid P \mid R_P \xrightarrow{\tau} ...$$

Obviously, for modeling, programming languages with higher-order constructs (such as functions with functions as parameters, or processes/agents that can migrate between hosts), the higher-order calculus is suitable. However, its theory is considerably complicated. Therefore, in some situations, it is advisable to encode it into the first-order calculus using the algorithm given by Sangiorgi in [89]. Instead of transmitting an agent P, we transmit a new name which can be used to trigger P. Since the receiver of P might invoke P several times (because the corresponding agent variable occurs at several places), P must be replicated. The main idea of the encoding $\| \bullet \|$ from higher-order to first-order calculus is as follows, where it is assumed that there exists a previously unused name y for each agent variable Y:

$$\begin{aligned} \|\bar{x}P.Q\| &= (\nu p)\bar{x}p.(\|Q\| \mid !p.\|P\|) \text{ where } p \notin fn(P,Q) \\ \|x(Y).P\| &= x(y).\|P\| \\ \|Y\| &= \bar{y} \end{aligned}$$

Let us consider the example above, $x(Y).(Y|Y) \mid \bar{x}Q.P \xrightarrow{\tau} Q|Q|P$, where Q and P contain no higher-order prefixes. Using the encoding, and assuming q is not free in Q or P, we get a similar behavior:

$$\begin{aligned} & x(y) \,.\, (\bar{y} \mid \bar{y}) \mid (\nu q)\bar{x}q \,.\, (P \mid !q \,.\, Q) \\ & \xrightarrow{\tau} (\nu q)(\bar{q} \mid \bar{q} \mid P \mid !q \,.\, Q) \\ & \xrightarrow{\tau}\xrightarrow{\tau} (\nu q)(0 \mid 0 \mid P \mid Q \mid Q \mid !q \,.\, Q) \\ & \equiv P \mid Q \mid Q \mid (\nu q)!q \,.\, Q \end{aligned}$$

where the component on the right hand-side $(\nu q)!q \,.\, Q$ will never be able to execute because it is guarded by a private name, so the whole term will behave as $Q \mid Q \mid P$ as expected.

Operational Semantics of the HOπ-calculus

The *operational semantics* are given in terms of a labeled transition system, where transitions are of kind $P \xrightarrow{\alpha}$ for some set of actions ranged over by α (see section 3.1.2). There are three labels for the transitions: the silent step τ, the input action $x\widetilde{K}$ and the output action $\overline{x}K$.
Output action: $\overline{x}K.P \xrightarrow{\bar{x}K} P$ means that after having sent message K (tuples of channels or processes) over channel x, process $\overline{x}K.P$ behaves like P.
Input Action: $x(K).P \xrightarrow{x(U)} P\{U/K\}$ means that if message U (tuples of channels or processes) is sent over channel x, then the process $x(K).P$, waiting for process or channel

names on x, receives it and replaces the process or channel names by U. It then behaves like P, where all occurrences of K are replaced by U. By sending an agent through a channel, real parameters are used, while (.) stands for formal parameters.
The interaction between two processes is given by the "Communication rules Com1 and Com2" in [90]:
$ComHO_1$:

$$\frac{P \xrightarrow{\bar{x}K} P', Q \xrightarrow{x(U)} Q'}{P|Q \xrightarrow{\tau} P'|Q'\{K/U\}}$$

$ComHO_2$:

$$\frac{P \xrightarrow{x(U)} P', Q \xrightarrow{\bar{x}K} Q'}{P|Q \xrightarrow{\tau} P'\{K/U\}|Q'}$$

The output action in $ComHO$ (see above) causes P to become P', the corresponding input action causes Q to become Q' then P and Q in parallel become P' and Q' in parallel. For the two rules of the operational semantics, as we have shown above, it is also necessary to notice that the amount of numbers of parameters of U is the same as the numbers of parameters of K: $\triangle_{A,P}(U) = \triangle_{A,P}(K)$ (see Def. 3.3) with A a set of actions $x, y, ...$; and P a set of agents $P, Q, ...$
Equivalences: Bisimulation usually identifies processes with the same external behavior. Higher-order bisimulation identifies higher-order processes if their interactions with the environment are the same and if their internal processes are bisimilar [69].

3.2 The Ambient-Calculus

"An ambient is a bounded place where computation happens. The interesting property is the existence of a boundary around an ambient. If we want to move computations easily we must be able to determine what should move; a boundary determines what is inside and what is outside an ambient"[63].
An ambient is a sort of a computation environment containing all the necessary data, code and processes. A whole ambient can move together with its whole content. The ambient calculus addresses also the problem of security (crossing the firewall).
The syntax of an ambient calculus is:

$$P, Q ::= (\nu n)P \mid 0 \mid P|Q \mid !P \mid M[P] \mid M.P \mid (x).P \mid < M >$$
$$M ::= x \mid n \mid in\, n \mid out\, n \mid open\, n \mid \epsilon \mid M.M'$$

Where

- n stands for names of ambients, ν is the restriction operator and $(\nu n)P$ creates a unique name n within a scope P
- 0 is the inactive process
- $|$ is the parallel operator
- ! is the replication operator
- $M[P]$ are ambients, $n[P]$ denotes an ambient of name n, and process P is running inside n
- $M.P$ are processes executing an action regulated by capability M and then continues as the process P
- (x) is an input action
- $< M >$ is an asynchronous output action that causes M to be put in the surrounding ambient where it can be caught by some process as input.
- M is a capability, it is either a variable x or an ambient name n. Mcan either be empty or a path of capabilities. x is an input action that causes process $(x).P$ to receive as input the name or capability x which is in its surrounding ambient.
- *in* for enabling an ambient to enter another ambient
- *out* for enabling an ambient to leave a surrounding ambient and let them become sibling ambients
- *open* for opening up an ambient (for dissolving and revealing the ambient and its content).

The operational semantics is sketched just for the capabilities an the communication primitives.

Entry Capability $n[in\, m.P|Q]|m[R] \stackrel{in\, m}{\rightarrow} m[n[P|Q]|R]$ means that the action $in\, m$ causes the surrounding ambient n to move to a sibling ambient m. As a result the whole ambient n (with all the processes inside) moves to ambient m.

Exit Capability $m[n[out\, m.P|Q]|R] \overset{out\, m}{\rightarrow} n[P|Q]|m[R]$ means that ambient n leaves ambient m and n and m become sibling ambients.

Open Capability $open\, n.P|n[Q] \overset{open\, n}{\rightarrow} P|Q$ means that ambient n is removed and the processes inside are revealed. The process that instructs the ambient to *open* is not in the ambient, but resides at the same level as the ambient.

Local anonymous communication $(x).P \mid < M > \rightarrow P\{x \leftarrow M\}$ means that the output action M is released in an ambient and that it is taken as input (in the same ambient)by a process. This process behaves after the input like P where x has been replaced by M. It is also possible to have communication between ambients (constructed on top of the communication inside an ambient).

Ambient reduction $\frac{P \rightarrow Q}{n[P] \rightarrow n[Q]}$ reflects the fact that a process executes inside an ambient. If the process executes in a given manner, then it will execute in the same manner inside the ambient.

An ambient can perform two different moves: a subjective and an objective one.
The *subjective move* happens when an ambient enters another ambient with all its processes inside after being instructed by a process inside this ambient to enter it. The exit capability is similar: a process inside the ambient instructs the ambient to exit its surrounding ambient.
An *objective move* enables a process inside an ambient to move from one ambient to another. The process that performs the objective "*in* move" enters a new ambient, but its surrounding ambient remains at its place. The process that performs the objective "*out* move" exits from the current ambient, the other processes remain in that ambient.
An ambient can be a *mobile process* which moves from one host (ambient) to another. An ambient can be a π-calculus channel which enables processes to communicate the names of other π-calculus channels. An ambient can be a firewall: a process that wants to cross a firewall has to know the password k (which is an ambient). The firewall itself is another abient w. A pilot process enters the ambient password k, and with the $in\, w$ capability enables the process to enter w provided the password k has been shown.

3.3 Mobile and Dynamic Petri Nets

The Petri nets are a formalism used to model concurrent and parallel systems. The original place/transition Petri nets have been extended in different ways in order to capture high-level abstractions which are difficult or impossible to express with pure

place/transition Petri nets.
The Mobile petri nets are introduced in [5]. Process mobility is expressed here using variables and colored tokens in an otherwise static net. Dynamic Petri nets extend mobile Petri nets with mechanism for modifying the structure of a Petri net, i.e., for creating new Petri nets when transitions are fired.
Mobile Petri nets are a variation of place/transition nets with colored tokens allowing names of places to appear as tokens. For instance, ready(PRINTER, TYPE), job(FILE,TYPE) → PRINTER(FILE) is a transition, whose pre- and post- conditions are the left and right part of the → symbol respectively. Capital names are variable names, they will be instantiated to actual names, only at the firing of the transition. There are two places involved in the pre-condition: *ready* and *job*, and one place in the post-condition $PRINTER$, a variable not known in advance. This transition means that if the spooler $PRINTER$ from type $TYPE$ is ready then the job of name $FILE$ and type $TYPE$ can be sent to the spooler $PRINTER$. The file $FILE$ has moved from the place job to the place $PRINTER$. The binding of variables at the firing time will determine which file be printed on which printer. The enabling and firing of transitions depends on a substitution function for the variables.
Mobile Petri nets handle mobility à la π-calculus, i.e., the mobile Petri net expresses the changing configuration of communication channels between processes. In mobile Petri nets, names of places are allowed to appear as tokens inside places. In π-calculus, names of channels are sent along channels.
The dynamic Petri nets are mobile Petri nets extended with a mechanism for creating new subnets when a transition is fired. This is achieved by allowing the postcondition of a transition to be an entire net and not only a set of places with a post-condition. The enabling and firing of dynamic Petri nets depends on a substitution function for variables. The firing of a transition first removes the unified token determined by the substitution function and pre-conditions of the transition, then adds new places and new transitions with pre- and post-conditions given by the subnets appearing in the post-conditions of the transition.
Mobile petri nets and π-calculus are equivalent in their way of specifying mobility, however they differ in the sense that Petri nets are well suited for modeling causality relations between events, while process algebra are well suited for composing processes.

3.4 Other Formalisms

The three formalisms introduced above represent a limited selection among many other approaches to formally describe concurrent and mobile systems. Modeling mobility,

both physical and logic, is an active subject of ongoing research.
The join-calculus [42] and the distributed join-calculus [34] are extensions of the π-calculus which introduce the notion of names location and distributed failure. Locations form a tree of embedded locations, and locations can move from one location to another. In the spi-calculus [1] security plays a bigger role; this π-calculus extension was designed for the description and analysis of cryptographic protocols.
An important aspect of all these formalisms is whether they can serve as a base for tools and for verification of properties. In [73] the π-calculus is modeled in terms of history-dependent automata (with local names in the transitions) that may lead to simpler and automatic validation procedures. Mobile UNITY [35], an extension of UNITY [25] that augments the program state with a location attribute and provides a programming notation for capturing mobility. Petri-nets based approaches called high-level Petri nets, were also developed in order to model mobility and mobile agent systems. The mobile Petri-nets [5], the dynamic Petri-nets [5] or the M-Nets [13] are such high-level nets.

3.5 Conclusion

This chapter presented briefly the two formalisms to express mobility. Serugendo et al. investigate in [65], several formalisms for concurrent systems which they applied to two mobile code environments Obliq [22] and Messengers [45]. The conclusion they came to, is that process algebra is more useful for highlighting the parallelism and choice aspect of processes. They classified the formalisms as follows:

Mobility with π-calculus used by the π-calculus and the mobile Petri nets. It is a mobility by reference passing. Processes do not move but the communication configuration changes.
Mobile Petri-nets and π-calculus are equivalent in their way of specifying mobility, however they differ in the sense that Petri nets are well suited for modeling causality relations between events, while process algebras are well suited for composing processes.

Mobility with Higher-Order π-calculus The processes can really be sent through channels, they can move and change their configuration. It is a "true" mobility.

Mobility with ambient calculus It is a more general kind of mobility as it allows mobility of processes, of channel names, and of a whole environment (a process with its surrounding context).

Trying to match the two formalisms described in this chapter with section 2.2.3, we can say that the first formalism, the π-calculus corresponds to a Remote Evaluation (REV) paradigm and the ambient calculus to the Code on Demand (CoD) paradigm.
We need a calculus which must match the last paradigm, the Mobile Agent (MA), in order to have a "real " mobility and to be able to model a mobile agent system. In the next chapter we will use the HOπ because this calculus is dedicated to the mobility and its matches very well the MA paradigm, we are interested in.

Part II

FROM SPECIFICATION TO VALIDATION

Chapter 4

Specification Part

Deriving a detailed design from informal requirements can be a tedious and error-prone endeavor unless a methodical and rigorous approach is used. When large systems of software are created, it is important that everything works as intended. An increasing number of designers are interested in scenario-driven approaches that allow them to focus on the main functional aspects of the system to be specified. One needs to describe the system and its behavior in a precise and formal way, in order to make it possible to assess the behavior of the system and either *prove* mathematically or *validate* through simulations, that the system does indeed what it should do. The best method to do this is to use both validation and verification. We present an approach where formal specifications are written within the process-algebraic language Higher-Order π-Calculus and some properties will be verified using the tool UPPAAL [12] (model-checking) and simulated using a tool developed for this research. We present the approach, which was published in [c], by using the Service Location Protocol (SLP) [52] as a case study.

4.1 Mobile feature of the SLP Protocol

An important area is mobile code for providing clients access to network services. By deploying mobile code for network service access, called service drivers, clients can download code for particular network services when needed exactly as capsules and mobile code allow routers and other infrastructure elements to download code for processing protocols that were not previously encountered.
Client application software interacts with the downloaded network service driver through

a standardized programmable interface defined for the service.
The Service Location Protocol (SLP) is an Internet Engineering Task Force (IETF) standard track protocol [31] that provides a framework to allow networking applications to discover the existence, location, and configuration of networked services in enterprise networks.
SLP can eliminate the need for the user to know the technical features of network hosts. With the SLP, the user needs only to know the description of the service he is interested in. Based on this description, SLP is then able to return the URL of the desired service. SLP is a language independent protocol. Thus the protocol specification can be implemented in any language. The SLP infrastructure consists of three types of agents:

1. *UserAgent (UA)* is a software entity that is looking for the location of one or more services,
2. *ServiceAgent (SA)* is a software entity that provides the location of one or more services,
3. *DirectoryAgent (DA)* is a software entity that acts as a centralized repository for service location information.

In order to be able to provide a framework for service location, SLP agents communicate with each other using eleven different types of messages (Figure 4.1). The dialog between agents is usually limited to very simple exchanges of request and reply messages. The syntax, semantics and scope of these messages are defined in [31].

- Service Request (SrvRqst)
  Message sent by UAs to SAs and DAs to lookup the location of a service.
- Service Reply (SrvRply)
  Message sent by SAs and DAs in reply to a SrvRqst. The SrvRply message contains the URL of the requested service.
- Service Registration (SrvReg)
  Message sent by SAs to DAs containing information about a service that is available.
- Service Unregister (SrvUnReg)
  Message sent by SAs to inform DAs that a service is no longer available.
- Service Acknowledge (SrvAck)
  A generic acknowledgment that is sent by DAs to SAs as a reply to SrvReg and SrvUnReg messages.

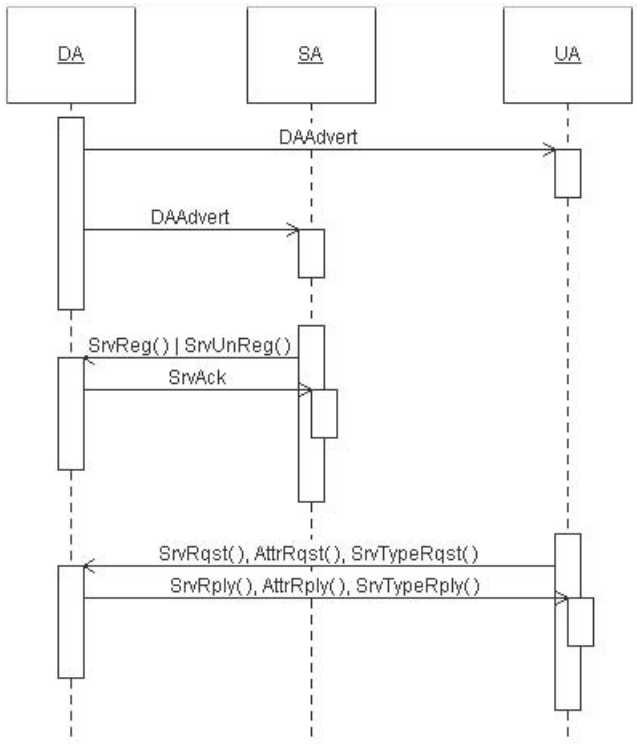


Figure 4.1: SLP Communications

- Attribute Request (AttrRqst)
  Message sent by UAs to request the attributes of a service.

- Attribute Reply (AttrRply)
  Message sent by SAs and DAs in reply to a AttrRqst. The AttrRply contains the list of attributes that were requested.

- Service Type Request (SrvTypeRqst)
  Message sent by UAs to SAs and DAs requesting the types of services that are available.

- Service Type Reply (SrvTypeRply)
  Message sent by SAs and DAs in reply to a SrvTypeRqst. The SrvTypeRply contains a list of requested service types.

- DA Advertisement (DAAdvert)
  Message sent by DAs to let SAs and UAs know where they are.

- SA Advertisement (SAAdvert)
  Message sent by SAs to let UAs know where they are.

SLP is a unicast and a multicast protocol. This means that the messages described above can be sent to one agent at a time (unicast) or simultaneously to all agents that are listening (multicast). A multicast is not a broadcast, because multicast messages are only "heard" by the nodes on the network that have "joined the multicast group". Multicast traffic from a given group is forwarded by routers to all subnets that have at least one machine that is interested in receiving the multicast for that group.
The minimal configuration of SLP requires two agent processes: the User Agent (UA) acts on behalf of a client to acquire service information, and the Service Agent (SA) acts on behalf of a service provider to disseminate information about the location and attributes of the service. In its most basic form, SLP is peer-to-peer. However, the initial service request is multicast to the SLP group address, since the UA does not know where the Service Agents are located. The UA must be prepared to receive multiple responses, since every SA within range that meets the service criteria of the request will respond. Starting the process of service discovery requires only the knowledge of a single well-known multicast group address, defined exclusively for SLP. Once a UA knows where and how to connect to a specific SA, subsequent requests are unicast directly to the SA (Figure 4.2). Thus, multicasting is limited to initial service discovery. All service

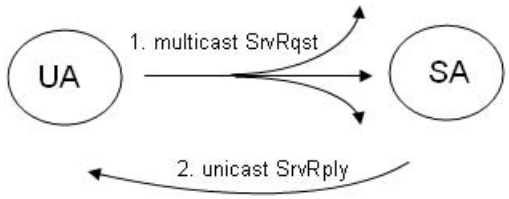


Figure 4.2: Service Discovery with UA and SA

requests include a predicate that is specified in terms of the service scheme attributes. The extended implementation of SLP includes a Directory Agent (DA), which acts as a centralized repository of service information - a kind of service switchboard. Service agents actively seek all directory agents using DA discovery; multicasting requests for the directory agent service type. A service registration is unicasted to each DA discovered. DAs actively advertise their service by multicasting advertisements. A user agent attempts to discover a DA initially. If successful, the UA unicasts service requests directly to the DA (Figure 4.3).
It is interesting to prove properties of applications to ensure that the developed codes

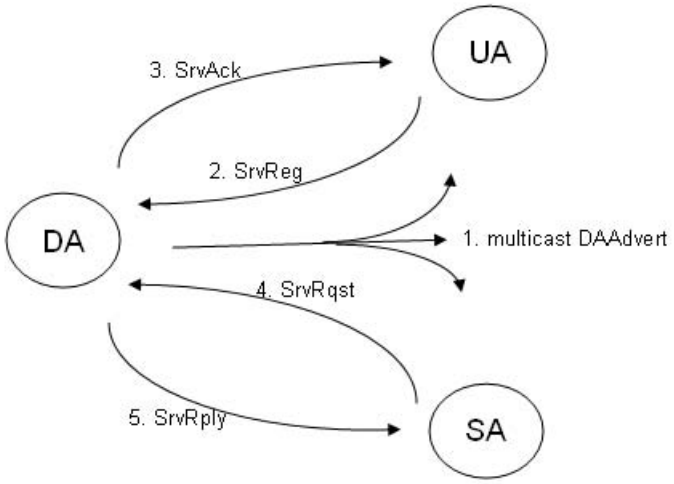


Figure 4.3: Service Discovery with UA, DA and SA

strictly adhere to their initial specifications. First, we look at a property of this protocol. This property must be expressed according to its initial specification, which also forms the base of the implementation of the SLP case study. Let us assume that a particular system (see figure 4.4) is composed of a *UserAgent (UA)*, a *ServiceAgent (SA)* and a *DirectoryAgent (DA)*. The *UserAgent* asks the *DirectoryAgent* for a service called "print", and the *DirectoryAgent* must be able to deliver a response. This interaction is one of many properties of this system. This property is basic and simple to understand, yet expressive. In order to prove properties of applications, we are interested in a formal specification language having the capacity to express mobility: the HOπ-Calculus.

4.2 Formal Specification of the SLP System

Our case study SLP [31], described above, is based on the publication of a simple print service. It uses just a single parameter. Also the Attr Messages (AttrRqst and AttrRply), the SrvType Messages (SrvTypeRply and SrvTypeRqst) will not be present in our specification, because they are treated as parameters of the higher-order terms in the HOπ (i.e. the information is an invariant within the requested property).
We give a formal specification of the SLP system using the HOπ-Calculus introduced in section 3.1.2, considering the particular scenario given above (see also section 4.1):

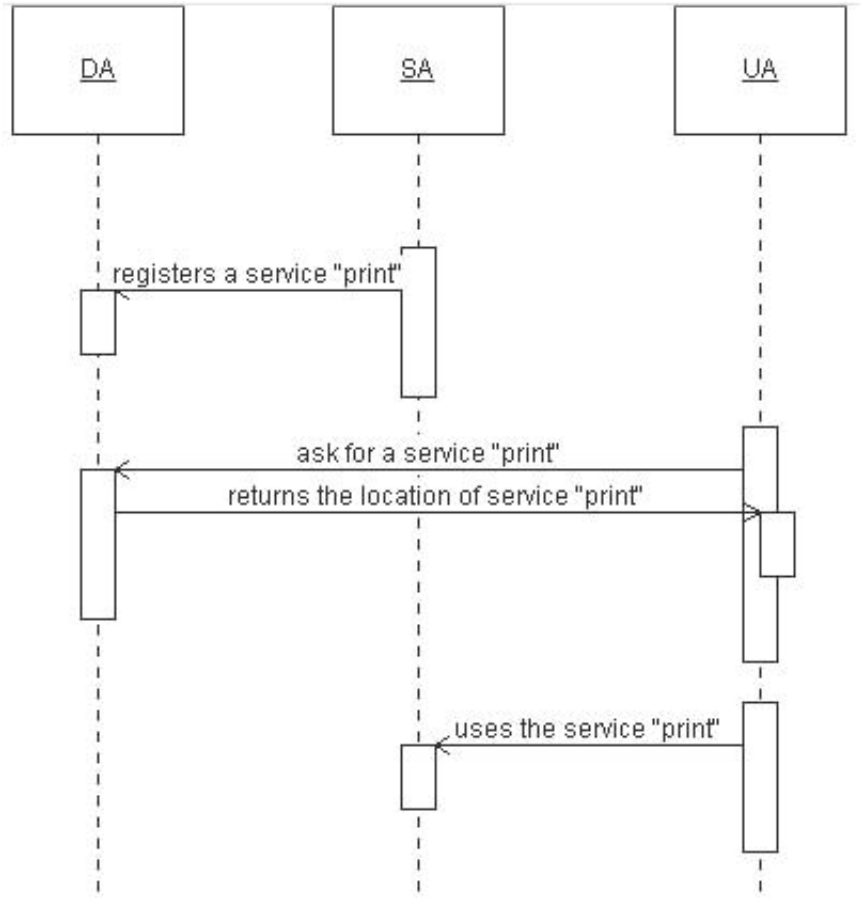


Figure 4.4: Sequence Diagram of our SLP Case Study

UserAgent (UA) asks the *DirectoryAgent (DA)* for a service called *print*. A *ServiceAgent (SA)* can register or unregister services with the *DirectoryAgent (DA)*, which as far as it is concerned must send the requested available service to the *UserAgent*. To simplify matters, our system is based on only one *SA*, one *DA* and one *UA*. We also assume that, through an unicast protocol, the *UA* knows the location of *DA* and *DA* knows the location of *SA*. Adopting this idea, we can ignore the Advert messages (DAAdvert, SAAdvert and UAAdvert), which are sent to let somebody know where they are located. We want to model the state of registered services, and because the specification language is a functional one, we define two agents called DA_{Mem} and $IdleDA_{Mem}$ for this purpose. Therefore, the two agents DA_{Mem} and the $IdleDA_{Mem}$ are used as a memory for a *DirectoryAgent (DA)*. The *DA* saves information in the DA_{Mem} and the DA_{Mem} in the

$IdleDA_{Mem}$ in order to be able to recover the information. The $IdleDA_{Mem}$ is also necessary to unregister services. In other words, we can say that DA_{Mem} models the information which is available on the DA, then the $IdleDA_{Mem}$ models the information which is already delivered to an agent.

Figure 4.5 shows the general working mechanism of the messages in a basic system that contains a *UserAgent*, a *ServiceAgent*, a *DirectoryAgent*, a Memory (DA_{Mem}) and an IdleMemory ($IdleDA_{Mem}$) for the *DirectoryAgent*. The formal specification of the SLP

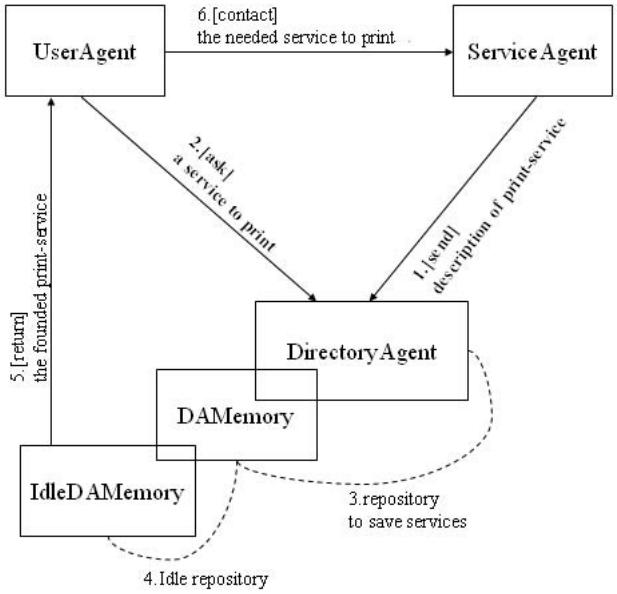


Figure 4.5: Messages between the agents in a basic system

system is given below:

System $= \nu(SrvRqst, SrvRply, SrvReg, SrvUnReg, SrvAck)$

$UA(SrvRqst, SrvRply)$

$|SA(SrvReg, SrvUnReg, SrvAck)$

$|DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)$

In this specification, the mobility is described through the parameters of the channel. The system is composed of one UA, one DA and one SA. For each new service

registered by the SA within the DA, there are a DA_{Mem} and a $IdleDA_{Mem}$ belonging to this specific service. In our case the service will be "Print". The parameters for each agent are used for the communication between agents.

$$\mathbf{UA(SrvRqst,\ SrvRply)} = \nu(Print)$$

$$\overline{SrvRqst}(Print, SrvRply)\,.\,SrvRply(Name).\,UA(SrvRqst, SrvRply)$$

We use gates like $SrvRply$ to specify the communication channel between agents. The UA specification is described as follows: the term $\overline{SrvRqst}(Print, SrvRply)$ is used in order to send to DA over the channel $SrvRqst$ an inquiry about an existing service "Print" and also the channel for further communication. The reception term $SrvRply$ reveals the higher-order aspect of this specification, because its scope is from the type agent.

$$\mathbf{SA(SrvReg, SrvUnReg, SrvAck)} = \nu(Print)$$

$$\overline{SrvReg}(Print)\,.\,SrvAck\,.\,(SA(SrvReg, SrvUnReg, SrvAck) + \overline{SrvUnReg}(Print)\,.\,SrvAck$$

$$.\,SA(SrvReg, SrvUnReg, SrvAck))$$

The SA can register a service within the DA using the channel $SrvReg$ and also remove a registered service within the DA using the channel $SrvUnReg$.

$$\mathbf{DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)} = \nu(input, reset, channel, inputIdle, resetIdle)$$

$$(SrvReg(S_{reg})\,.\,((DA_{Mem}(input, reset, channel, inputIdle)$$

$$|\,IdleDA_{Mem}(input, resetIdle, channel, inputIdle)\,|\,\overline{input}(S_{reg})\,.\,\overline{SrvAck})$$

$$+\,SrvUnReg(S_{reg})\,.\,\overline{reset}\,.\,\overline{resetIdle}\,.\,\overline{SrvAck})$$

$$|\,SrvRqst(S_{rqst}, ch)\,.\,\overline{channel}(S_{rqst}, ch))$$

$$.\,DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)$$

The *DA* can register or unregister a service. After having registered a service, the *DA* will start a DA_{Mem} and an $IdleDA_{Mem}$ and will send the information to the memory DA_{Mem}. Along with the request for the service "Print", the *DA* receives the channel name *SrvRply*. This channel is used later by $IdleDA_{Mem}$ to send the requested service back to the *UA*.

DA_{Mem}(input, reset, channel, inputIdle) =

$input(S_{reg}) \,.\, channel(S_{rqst}, ch) \,.\, [S_{reg} = S_{rqst}]\, \overline{ch}(S_{reg}) \,.$

$\overline{inputIdle}(S_{reg}) \,.\, DA_{Mem}(input, reset, channel, inputIdle) \,+\, reset.0$

After having initially received the first time the registered service over the channel *input* and the channel *SrvRply* over the channel *channel* from *DA*, the DA_{Mem} sends the requested service to *UA*, as long as it matches with the registered service. To be able to send the service "Print" several times, the DA_{Mem} will store the service and channel information in the extra memory named $IdleDA_{Mem}$.

IdleDA$_{Mem}$(input, resetIdle, channel, inputIdle) =

$inputIdle(S_{reg}) \,.\, channel(S_{rqst}, ch) \,.\, [S_{reg} = S_{rqst}]\, \overline{ch}(S_{reg}) \,.$

$\overline{input}(S_{reg}) \,.\, IdleDA_{Mem}(input, resetIdle, channel, inputIdle) \,+\, resetIdle.0$

The $IdleDA_{Mem}$ satisfies the *UA*'s request by sending the correct service. This answer is done on the *SrvRply* channel (which substitutes *ch*). In any case the system can be modeled by several *ServiceAgents* and several *DirectoryAgents*, depending on the number of services present in the network. The location of these agents is then known through a multicast request. The *UserAgent* can also contact several *ServiceAgents*. For instance, if it searches two services, a print service and a mail service, one of which is registered with a $DirectoryAgent_1$ and the second with a $DirectoryAgent_2$ the *UserAgent* can exchange data with both of them. There is also a need to provide for each registered service a proprietary DA_{Mem} and an $IdleDA_{Mem}$.
Formal methods may be used to specify and model the behavior of a system and to verify that the design and implemented system satisfy the expected functional and safety properties. The higher-order π-calculus is able to describe the behavior of mobile sys-

tems and, because it possesses formal semantics, it is capable of verification as well. With the notion of transition graphs and the operational semantics for operators in the HOπ-Calculus, we can demonstrate the relation between them. The operational rules will allow us to prove or disprove the correctness of the arcs connecting agent expressions in any transition graphs we construct. We do this by giving a directive (see Figure 4.6) on how to construct inference trees [37] using just the rules we have stated for the operational semantics. At the root of each successful tree will be the transition we are trying to prove. Each node in the tree will consist of a transition labeled by the rule which was used to derive it. A node may only refer to transitions already proved correct higher up in the tree. The treatment of non-determinism in the specification of our case study is represented by offering the choice of a channel *reset*, which is needed in order to remove the registered services.

The construction of such a tree must start with the principal term which is *System* in our case. We consider the same case study, with one exception: the UA will only request the service "Print" once. With this consideration, the requested service will be returned by the DA_{Mem} and the $IdleDA_{Mem}$ will not be needed. The function of the $IdleDA_{Mem}$, being the equivalent of the DA_{Mem}, will not be discussed in detail. Furthermore we define and prove that the reachability property (*If UA asks for the service "print" and if SA subscribes a DA, then UA obtains the "print" service*), has been achieved.

Using the communication rule $ComHO_1$ and $ComHO_2$ between two processes we can establish the correctness of our property. The communication rules can only be used if the interacting agents have the same arity, as we demand in the first section of this chapter.

STEP 1 in the construction of our inference tree consists of the interaction between the UA and the DA (meaning that UA *asks* the DA for the service "print"). At the same time, the UA communicates to the DA its location and the name of the communication channel (in this case $SrvRply$). The interacting channels are written in **bold**.

$UA(SrvRqst, SrvRply) = \nu(Print)$
$\overline{\mathbf{SrvRqst}}\mathbf{(Print, SrvRply)} \,.\, SrvRply(Name).UA(SrvRqst, SrvRply)$

$DA(SrvReg, SrvUnReg, SrvRqst, SrvAck) = \nu(input, reset, channel, inputIdle, resetIdle)$
$(SrvReg(S_{Reg}) \,.\, ((DA_{Mem}(input, reset, channel, inputIdle)$
$\mid IdleDA_{Mem}(input, resetIdle, channel, inputIdle) \mid \overline{input}(S_{reg}) \,.\, \overline{SrvAck})$
$+ SrvUnReg(S_{reg}) \,.\, \overline{reset} \,.\, \overline{resetIdle} \,.\, \overline{SrvAck})$

$\mid \mathbf{SrvRqst(S_{rqst}, ch)} \,.\, \overline{channel}(S_{rqst}, ch)) \,.\, DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)$

STEP 1

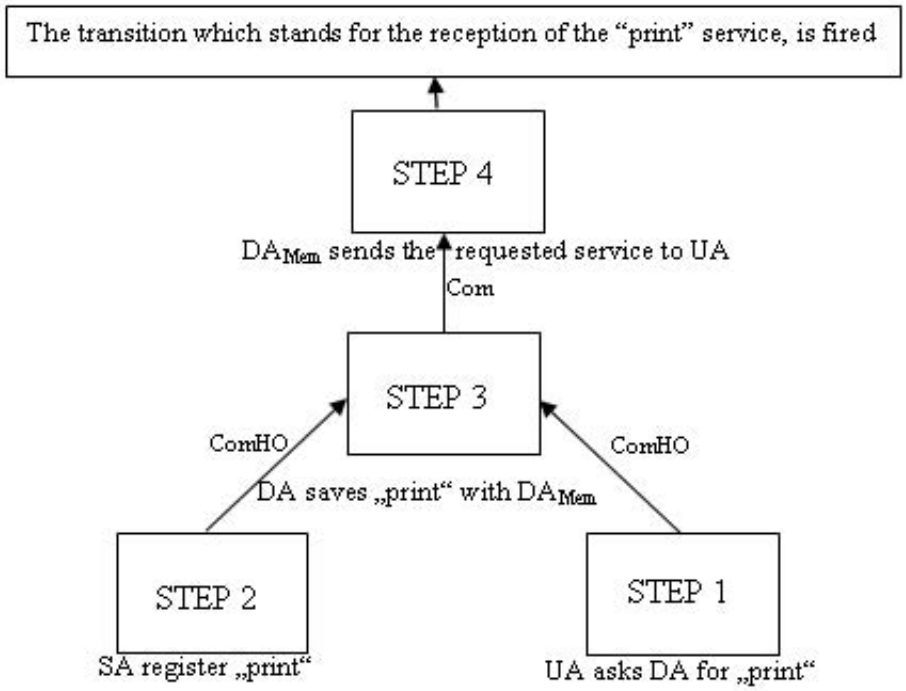


Figure 4.6: How to construct the inference tree

STEP 2 consists of the interaction between the SA and the DA (meaning that SA *subscribes* the service "Print" with the DA). The interacting channels are written in **bold**.

$$
\begin{aligned}
&DA(SrvReg, SrvUnReg, SrvRqst, SrvAck) = \nu(input, reset, channel, inputIdle, resetIdle)\\
&\mathbf{(SrvReg(S_{Reg})}\,.\,((DA_{Mem}(input, reset, channel, inputIdle)\\
&\mid IdleDA_{Mem}(input, resetIdle, channel, inputIdle) \mid \overline{input}(S_{reg})\,.\,\overline{SrvAck})\\
&+ SrvUnReg(S_{reg})\,.\,\overline{reset}\,.\,\overline{resetIdle}\,.\,\overline{SrvAck})\\
&\mid SrvRqst(S_{rqst}, ch)\,.\,\overline{channel}(S_{rqst}, ch))\,.\,DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)\\
&\\
&SA(SrvReg, SrvUnReg, SrvAck) = \nu(Print)\\
&\mathbf{\overline{SrvReg}(Print)}\,.\,SrvAck\,.\,(SA(SrvReg, SrvUnReg, SrvAck)\\
&\\
&+ \overline{SrvUnReg}(Print)\,.\,SrvAck\,.\,SA(SrvReg, SrvUnReg, SrvAck))
\end{aligned}
$$

STEP 2

According to the application of the $Coms$ rules for the parallel communication between the channels written in bold in STEP 1 and STEP 2, the DA and DA_{Mem} communi-

cate together through the channels *input* and *channel*. Because of the symmetry of the semantics of the operator |, we can begin the construction of our tree either with STEP 1 or with STEP 2.

The system in **STEP 3** becomes:

$DA(SrvReg, SrvUnReg, SrvRqst, SrvAck) = \nu(input, reset, channel, inputIdle, resetIdle)$
$(SrvReg(S_{Reg}) . ((DA_{Mem}(input, reset, channel, inputIdle)$
$| IdleDA_{Mem}(input, resetIdle, channel, inputIdle) | \mathbf{\overline{input}(S_{reg})} . \overline{SrvAck})$
$+ SrvUnReg(S_{reg}) . \overline{reset} . \overline{resetIdle} . \overline{SrvAck})$
$| SrvRqst(S_{rqst}, ch) . \mathbf{\overline{channel}(S_{rqst}, ch)}) . DA(SrvReg, SrvUnReg, SrvRqst, SrvAck)$

$DA_{Mem}(input, reset, channel, inputIdle) =$
$\mathbf{input(S_{reg})} . \mathbf{channel(S_{rqst},\ ch)} . [S_{reg} = S_{rqst}] \overline{ch}(S_{reg}) .$

$\overline{inputIdle}(S_{reg}) . DA_{Mem}(input, reset, channel, inputIdle) + reset.0$

STEP 3

The communication between DA and DA_{Mem} through the mutual channel *input* means that DA sends this information (that the registered service is "print") to its memory DA_{Mem}. This communicates with $IdleDA_{Mem}$ in order to save the information. We need this second "saving element" in order to be able to recover the information when the SA wants to unregister it or the UA desires the same service several times.

In **STEP 4** when the UA asks for the first time for a service "Print", the DA_{Mem} will return the desired service using the channel *SrvRply*

$UA(SrvRqst, SrvRply) = \nu(Print)$
$\overline{SrvRqst}(Print, SrvRply) . \mathbf{SrvRply(Name)} . UA(SrvRqst, SrvRply)$

$DA_{Mem}(input, reset, channel, inputIdle) =$
$input(S_{reg}) . channel(S_{rqst}, ch) . [\mathbf{S_{reg}{=}S_{rqst}}] \mathbf{\overline{ch}(S_{reg})} .$

$\overline{inputIdle}(S_{reg}) . DA_{Mem}(input, reset, channel, inputIdle) + reset.0$

STEP 4

The channel *ch* became *SrvRply* after its substitution. If the registered and the requested service are the same, the DA_{Mem} will send it to the UA. After the communication, the UA receives the requested service "Print".

Our specification contains a second memory, called $IdleDA_{Mem}$, used in the case the UA asks a second time, the same service or the service has to be removed. The DA_{Mem} registered the service "Print" with the $IdleDA_{Mem}$. After all communications and application of $ComHO$ has been terminated, the system becomes:

$UA|SA|DA|DA_{Mem}|IdleDA_{Mem}$ with $Name := Print$. This result means that the last transition is achieved and the *UserAgent* can use the requested service "Print".

4.3 Temporal Properties of SLP Protocol

Model checking allows the discovery of well hidden errors in a large system. For example, with the help of the verification tool UPPAAL [12], an error in an audio-video control protocol from Bang & Olufsen has been discovered and a correction proposal has been made. In order to verify automatically a system using a model checking method, it is necessary to first build a formal model (e.g. an automaton). To do this, one needs to employ a system specification language. Therefore, formal methods are typically used for the specification of agent systems and agent behaviors. The primary purpose of the resulting formal agent model is to define which properties are to be realized by the agent system, e.g. behavioral properties. To prove behavioral properties of our SLP protocol, we need a properties specification language as, for instance, a temporal logic.
Model checking is a technique in which the verification of a system is carried out by using a finite representation of its state space. Basic properties, such as an absence of deadlock or satisfaction of a state invariant (e.g. mutual exclusion), can be verified by checking individual states. More subtle properties, such as guarantee of progress, require checking for specific cycles in a graph representing the states and possible transitions between them. Properties to be checked are typically described by formulae in a branching time or linear time temporal logic.
There are several model checkers that we could use in order to prove system properties. Three of the most well-known are the HD-Automata Laboratory (Hal) environment [38], the mobility Workbench [99] for analyzing π-calculus processes and the UPPAAL [12] that uses the computation tree logic for the verification of a property. None of them handle higher-order π-calculus, therefore our case study must be converted into a first-order calculus (see the translation given by [89]). The HAL is an integrated tool set for the specification, verification and analysis of concurrent and distributed systems. The core of HAL consists of the HD-automata: they are used as a common format for the various history-dependant languages. The HAL environment includes modules which implement decision procedures to calculate behavioral equivalences, and modules which support verification of behavioral properties expressed as formulae of suitable temporal logics. At this moment HAL works only with concurrent and distributed systems expressed by a basic π-calculus formalism. The HAL environment allows π-calculus agents to be translated into ordinary automata, so that existing equivalence checkers can be used to calculate whether the π-calculus is bisimilar. The environment also supports ver-

ification of logical formulae expressing desired properties of the behavior of π-calculus agents. The tool is still a prototype and currently only works under Unix or online [`http://fmt.isti.cnr.it:8080/hal/bin/HALOnLine/`]. For the online version, there is no debugger or syntax checker, which makes it difficult to enter the correct data. This tool does not allow either the expression of matching a polyadic term or of a higher-order one. The deficient visualization of the automata convinced us to use the UPPAAL tool which is a good tool for the modeling and verification of concurrent systems which communicate through synchronization. The tool has a user-friendly interface, is easy to use and also provides a simulation function. This function is the strength of this tool, because it allows the manipulation and the observation of the system behavior. It even delivers useful diagnosis for a large system, which cannot use verification. UPPAAL also permits the simulation of unpredictable performances: at every stage where several transitions are possible, one of them is haphazardly chosen. Finally, during the verification of a property, it can acquire a diagnosis in the form of an execution leading to the sought-after configuration. This sequence can be memorized to then be played again by the simulator. Even though the simulation does not permit the exhaustive examination of a system's behavior, it is an important phase in order to have confidence in the constructed model.

In order to verify properties, the tool allows us to state them in the form of Computation Tree Logic (CTL) formulae. CTL is a branching time [32] temporal logic which provides correct behavior of parallel systems by expressing properties concerning the occurrence of events in time. Different operators and modalities can be used to express important properties such as invariance, eventuality and precedence [33]. Given a correctness property, i.e. a temporal logic formula, there are two principal ways of using temporal logic. One is to apply an automated synthesis method using a decision procedure to determine the satisfiability of our property. When the method succeeds, it generates a synchronization skeleton of the system events [27]. The second one, on which we will focus our attention, uses a model checking based method, which verifies the truth of the correctness property in the structure representing the parallel system. This structure is generally a labeled transition system.

Neither HAL, nor UPPAAL are able to represent a higher-order π-calculus term. In order to represent this kind of term, we have to transform them into a first-order π-calculus. A transformation algorithm is given by Sangiorgi in [89]. Following his rules for the compilation from HOπ to π-calculus we can proceed to the transformation of our initial HOπ specification into a first-order, one which is necessary if we wish to use a model checker such as HAL or UPPAAL. No existing model checker can operate on a HOπ term.

We decided to prove our properties using the UPPAAL tool, because it is a known,

working tool and not a prototype and we could try it under different platforms. It uses diagrams for the representation of the agents, making it more convenient and easier to proceed; it provides diagnostic information and is relatively easy to learn and use. UPPAAL is an integrated tool environment for modeling, simulation and verification of real-time systems, developed jointly by Basic Research in Computer Science at Aalborg University in Denmark and the Department of Information Technology at Uppsala University in Sweden. A typical application area includes the communication protocols. This was one of the reasons for choosing this tool. It is designed mainly to check invariant and reachability properties by exploring the state-space of a system. A system in UPPAAL is composed of concurrent processes, each of them modeled as an automaton. The automaton has a set of locations. Transitions are used to change location. To control how to fire these transitions, it is possible to have a guard and a synchronization. A guard is a condition on the variables stating whether or not the transition is enabled. The synchronization mechanism in UPPAAL is a hand-shaking synchronization: two processes fire a transition at the same time, one will have $a!$ (send) and the other $a?$ (receive), a being the synchronization channel. Actions are possible by taking a transition: assignment of variables.
UPPAAL can be used to verify properties on (real time) systems:

Safety Safety properties are some properties that are required to always hold, i.e. they are invariants.
- $A[\,]\varphi$ means that *"for all paths φ will always hold"*.
- $E <> \varphi$ means that *"for some paths φ will eventually hold"*.

Reachability A reachable property is a property that can eventually hold.
- $A <> \varphi$ means that *"for all paths φ will eventually hold"*.
- $E[\,]\varphi$ means that *"for some paths φ will always hold"*.

Bounded liveness Bounded liveness properties are properties that have to hold within a certain time bound.
- $F ::= F\ U_{<t}\ P$ means that *"F holds until P and P holds before t"* where F and P are properties (logical expressions) and t is time.

Let us define for our SLP system from the section 4.1, for each category of properties, a specific property:

- Safety: *"Every agent SA which registers a service within DA is recognized by the DA and the service becomes available"*.
- Reachability: *"If UA asks for the service "print" and if SA subscribes a DA, then UA obtains the "print" service"*.

- Bounded liveness: *"During the registration period within the DA, a service must be accessible (after SrvReg and before SrvUnReg)".*

UPPAAL's property specification language is actually a subset TCTL (Timed Computation Tree Logic) because it does not allow $A[\,]$, $E<>$, $A<>$ and $E[\,]$ to contain one another. Unlike in complete TCTL these can only be written before a logical expression. Using this notation, we can specify and verify some key properties of our SLP protocol, shown above, such as reachability and safeness.

4.3.1 SLP modeled by UPPAAL

The aim of this section is to verify if a given property is satisfied. We consider our SLP system from section 4.2 and we consider that the UA only requests the service "Print" once. This consideration will eliminate the need of a second memory *DirectoryIdleMem* (which is used only if UA requests the same service several times). The system is modeled by UPPAAL through a set of automata using exactly the same names for communication channels. There are some constraints that we have to consider. The first one is that the initial $HO\pi$ specification must be transformed into a first-order one. We do not use Sangiorgi's algorithm for the compilation, because we do not need it for such simple systems. We just consider the service "Print" as a name. Because "Print" is the only agent in our system, this transformation is sufficient. Another constraint is that the system does not allow the matching of names, we must also consider that the registered service corresponds to the requested one. Because of the weak power of expression of UPPAAL, the specification describes, therefore, only the request of a service "Print".
The textual description of a system (in Source Code 3) begins with the declaration of variables, constants and communication channels. The system defines two constants *Print* for the requested and registered service and *Unknown* for an unknown service. It defines also a global clock and three more constants fast, slow, slowest in order to define which agent is the fastest. SA is the fastest agent, the UA the slow one and the other agents even slower. That means that the SA is the first agent to register and only after that UA can ask for its service. It also defines the channels responsible for the synchronic communication between agents and the variable "printcount" in order to show that the UA can request the same service up to ten times. The variables defined on line 4 are communication assignments.

Source Code 3 — *Configuration of SLP System*

```
1:  const Print 1;
```

```
2:  const Unknown 0;
3:  chan printRply, printRqst, printReg, printUnReg, ack, input, reset,
    resetIdle, channel, inputIdle;
4:  int[0,1] requested_service, registered_service, unregistered_service;
5:  int[0,10] printcount;
6:  clock global;
7:  const fast 5;
8:  const slow 10;
9:  const slowest 15;
```

The specification of the UA process defines two communication channels. We model them using three control states one between each action. The synchronization action *sync* permits us to model the communication. The definition of *UserAgent* begins with the list of the control states. We consider systematically that every action (belonging to α, i.e. receiving, sending or the silent action) has a time duration equal to zero, but there exists a permanent control state before and after the action. After the definition of the initial state *start_UserAgent*, we give the transition list, the control states (start and end), the synchronization action and the variable assignments. The textual description of the *UserAgent* is given below in Source Code 4.

Source Code 4 — *Process UserAgent*

```
1: process UserAgent
2: {
3:   states: start_UserAgent, request_service_Print,
   service_Print_is_received;
4:   start_UserAgent -> request_service_Print
5:   {
6:    assign requested_service:=Unknown, clockUA:=0;
7:    sync printRqst!; //This transition waits for a synchronization in
   order to send its request.
8:   },
9:   request_service_Print -> service_Print_is_received
10:  {
11:   guard clockUA > delay;
```

```
12:    assign requested_service:=Print;
13:    sync printRply?; //This transition waits for a synchronization in
    order to get its requested service.
14:  },
15:  service_Print_is_received -> start_UserAgent
16: }
```

Instead of a textual description of the system we can also use the graphic editor from UPPAAL to describe the different automata to define variables, etc... The figures below were created using the latest version of UPPAAL 3.4.8. The modeling of *UserAgent* is given by Figure 4.7 and describes the states when the *UserAgent* sends a request concerning a "Print" service. The agent has three locations: one to start the agent *start_UserAgent* (the start location is marked by a double circle), the second *request_service_Print* which shows the state before and after its request for the service "Print" and the third *service_Print_is_received* before and after the service "Print" is received.

The specification of process *ServiceAgent* defines four communication channels, mod-

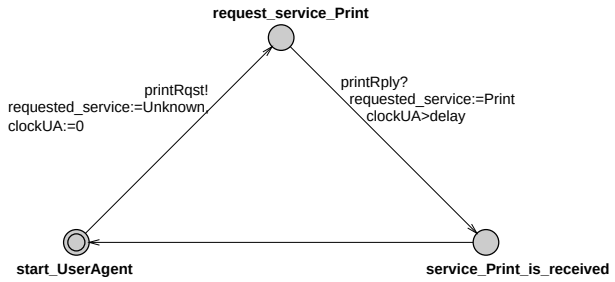


Figure 4.7: Modeling of UserAgent with UPPAALTool: process UA

eled by UPPAAL through four control states. The textual description of the *ServiceAgent* below in Source Code 5, shows the process of registering and unregistering services with the DA. After registering or unregistering an acknowledgment is sent. The process is also modeled in Figure 4.8.

Source Code 5 — *Process ServiceAgent*

```
1: process ServiceAgent
```

```
2: {
3:   states: start_ServiceAgent, register_service_Print,
   acknowledgement_sent; unregister_service_Print;
4:   start_ServiceAgent -> register_service_Print
5:   {
6:    assign clockSA:=0;
7:    sync printReg!;
8:   },
9:   register_service_Print -> acknowledgement_sent
10:  {
11:   guard clockSA > delay;
12:   sync ack?;
13:  },
14:  acknowledgement_sent -> unregister_service_Print
15:  {
16:   assign clockSA:=0;
17:   sync printUnReg!;
18:  },
19:  unregister_service_Print -> start_ServiceAgent
20:  {
21:   sync ack?;
22:   guard clockSA > delay;
23:  };
24: }
```

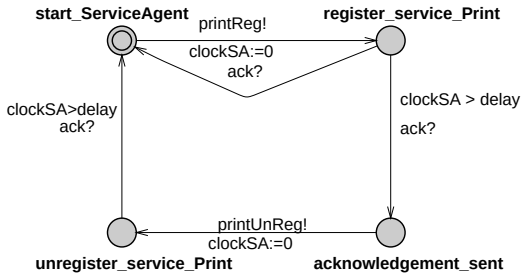


Figure 4.8: Modeling of Service Agent with UPPAAL: process SA

The *DirectoryAgent* is divided in two automata: *DirectoryAgentReg* and *DirectoryAgentSave*, as every possible communication has to be taken into account, and we can have two: one describing the registering and unregistering of services, the second describing the communication with the UA and the sending of the desired service. As in the first two processes, the textual descriptions follow the given π-calculus specification. The textual description of the *DirectoryAgentReg* is given below in Source Code 6.

Source Code 6 — *Process DirectoryAgentReg*

```
process DirectoryAgentReg
{
  states: start_DirectoryAgentRegister, service_Print_is_registered,
  save_service_Print, send_acknowledgement,
  service_Print_is_unregistered, delete_service_Print_from_Memory,
  delete_service_Print_from_IdleMemory;
  start_DirectoryAgentRegister -> service_Print_is_registered
  {
    assign registered_service:=Print, clockDAReg:=0;
    sync printReg?;
  },
  service_Print_is_registered -> save_service_Print
  {
    guard clockDAReg > delay;
    sync input!;
  },
  save_service_Print -> start_DirectoryAgentRegister
  {
    sync ack!;
  }//The system can do this transition and go back to the initial
  state or the next one in order to unregister the service.
  save_service_Print-> send_acknowledgement
  {
    sync ack!; // Using this transition an acknowledgement will be sent
    after the service "Print" is saved.
    guard clockDAReg:=0;
  },
```

```
23:  send_acknowledgement -> service_Print_is_unregistered
24:  {
25:   guard clockDAReg > delay;
26:   assign unregistered_service:=Print;
27:   sync printUnReg?;
28:  },
29:  service_Print_is_unregistered -> delete_service_Print_from_Memory
30:  {
31:   sync reset!; //This transition is used in order to delete the
    service "Print" from the Memory.
32:  },
33:  delete_service_Print_from_Memory ->
    delete_service_Print_from_IdleMemory
34:  {
35:   sync resetIdle!; //This transition is used in order to delete the
    service "Print" from the IdleMemory.
36:  };
37:  delete_service_Print_from_IdleMemory -> start_DirectoryAgentRegister
38:  {
39:   assign clockDAReg:=0;
40:   sync ack!;
41:  };
42: }
```

The graphical *DA* is described by Figure 4.9 and Figure 4.10. The first schema 4.9 models the process of registering and unregistering the services. The textual description below in Source Code 7 and the second schema 4.10 depicts the process of memorizing services in order to be able to reply to requests.

Source Code 7 — *Process DirectoryAgentSave*

```
1: process DirectoryAgentSave
2. {
3:   states: start_DAsave, request_for_Print_service_received;
4:   start_DAsave -> request_for_Print_service_received
```

```
5:   {
6:    assign clockDASave:=0;
7:    sync printRqst?;
8:   },
9:   request_for_Print_service_received -> start_DAsave
10:  {
11:   guard clockDASave > delay;
12:   sync channel!; //This transition is used in order to save the
    requested service.
13:  };
14: }
```

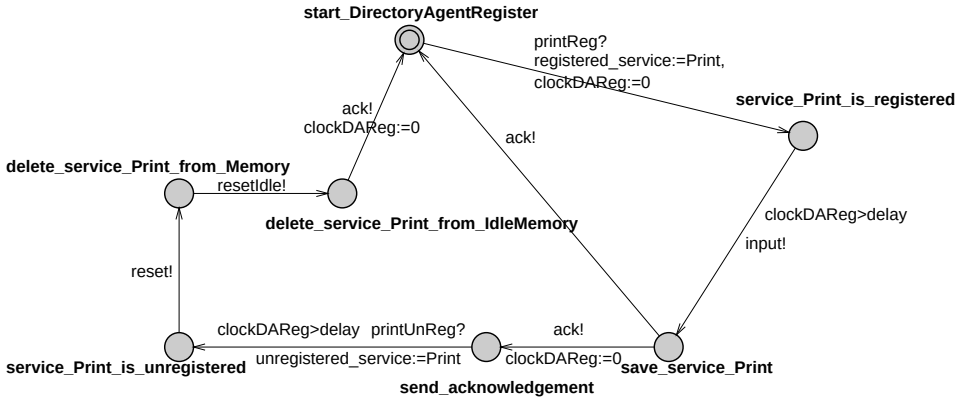


Figure 4.9: Modeling of DirectoryAgent with UPPAAL: the process of registration and unregistration a service

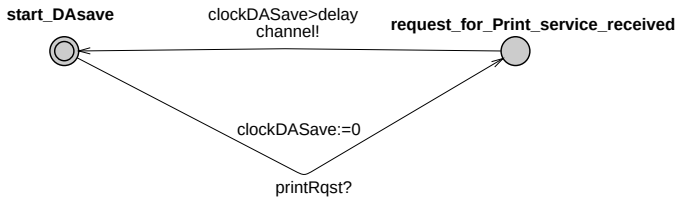


Figure 4.10: Modeling of DirectoryAgent with UPPAAL: the process of memorizing a service

The last figure 4.11 and the textual description below in Source Code 8 show the memory of the *DirectoryAgent*, the *DirectoryAgentMem* which is introduced as a container for saving information such as services. To be able to recover or delete these services we also use another container called *IdleDirectoryAgentMem*. The schema of this container is identical to the *DirectoryAgentMem* schema; it uses only its own variables, which are defined within the formal specification. If the UA requests several different services, the DA has to start one $DirectoryAgentMem$ and one $IdleDirectoryAgentMem$ for each service. The expression power of CTL forces the system to be constant and does not permit the dynamic evolution of the number of the saved services.
The entire system, which we depict using UPPAAL, respects the π-Calculus formal specification given above.

Source Code 8 — *Process DirectoryAgentMem*

```
1: process DAMem
2. {
3:   states: start_DAMem, service_Print_is_saved,
   request_for_service_Print_is_saved, send_requested_service_Print;
4:   start_DAMem -> start_DAMem
5:   {
6:   sync reset?;
7:   }, //This transition or the next one can fire first.
8:   start_DAMem -> service_Print_is_saved
9:   {
10:   assign registered_service:=Print, clockDAMem:=0;
11:   sync input?;
12:  },
13:   service_Print_is_saved -> request_for_service_Print_is_saved
14:  {
15:   assign printcount:=printcount+1; //After each request for the same
   service, the variable printcount increases.
16:   sync channel?;
17:  },
18:  request_for_service_Print_is_saved -> send_requested_service_Print
19:  {
```

```
20:    guard clockDAMem > delay;
21:    sync printRply!;
22:   },
23:   send_requested_service_Print -> start_DAMem
24:   {
25:    sync inputIdle!;
26:    assign clockDAMem:=0;
27:   },//This transition or the next one can fire first.
28:   send_requested_service_Print -> start_DAMem
29:   {
30:    sync reset!;
31:   };
32: }
```

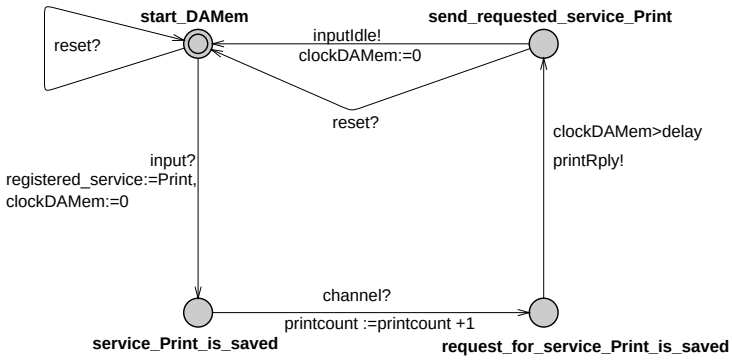


Figure 4.11: Modeling of DirectoryAgentMem with UPPAAL: process DAMem

Finally, the system, our network from section 4.1, is realized by an instruction of the type *system UA, SA, DAReg, DASave, DAMem, IdleDAMem.*

While we build a model, the simulation is one of the most practical approaches to reveal its validation. Simulation allows us, with the help of a graphical interface, to represent a system configuration with its different possible transitions. In a "step-by-step" simulation, we can choose a particular transition, get a new configuration and restart it. This allows us to manipulate the automata and observe its behavior. This sequence can be memorized in order to be played again by the simulator. In any case, we have to remember that the simulation does not comprehensively examine the behavior of a

system, and it is not possible to definitively conclude whether the system is correct or not at this stage.
The next step is to verify if a given property is satisfied using the model checker from UPPAAL.
$S.r$ expresses that the automaton S is in state r.
The property is defined: *"If UA requests a service "Print", the DirectoryAgentMem or the DirectoryAgentIdleMem is able to save the requested service"*. The first formula P_0 states that, in the reachability graph of SLP model, for all paths the states will always hold in the next state:
$P_0 = A <> ((UA.request\_service\_Print \; or \; DASave.request\_for\_Print\_service\_received)$ *imply*
$(DAMem.request\_for\_service\_Print\_is\_saved \; or \; IdleDAMem.service\_Print\_is\_saved))$
UPPAAL evaluates that the first formula P_0 is true.
The states will only eventually hold because the $DirectoryAgentIdleMem$ will save the service only if UA requests the same service at least twice.
The second formula P_1 establishes that in the reachability graph of our SLP model there exists a path where:
$P_1 = E[\,]((DAReg.service\_Print\_is\_registered \; and \; UA.request\_service\_Print)$ *imply* $(DAMem.send\_requested\_service\_Print))$ holds in the next state. This property expresses the case when the SA has already registered a service "Print", and the UA requests a service "Print" then the DA_{Mem} will deliver this service to UA. UPPAAL evaluates that the second formula P_1 is true. Consequently, the two properties are satisfied.

4.3.2 Conclusion

This section discusses how to specify a real protocol with mobile features, called SLP and also how to verify liveness properties of this protocol by using an existing verification tool called UPPAAL.

We have encountered two main limits. The first one occurred with the use of the π-Calculus language, its operational semantics and the tree inference construction. Model checking techniques can be used to prove some behavioral properties such as safety, reachability or correctness of the inheritance mechanism. We give the directive idea how to built an inference tree for proving that a transition "print" was achieved. This approach is not automatic and it is essential to explore the whole algebraic term before concluding that a specific state has been reached. This kind of proof is too specific. We last looked at an evaluation of a global term and cannot conclude anything about the

other possible evaluations. Moreover, our inference tree describes one UA, one SA and one DA. However, if we change the initial configuration, this tree must be built again from scratch.

Thus we used a model checking technique and temporal logic to establish a more general property about all the evaluations of a term. We have used the UPPAAL tool to accomplish this, yet we encountered a second limit; a sequence of actions is described by an automaton, yet between two automata there is no exchange of data. There is only one possible operation: the synchronization. In that case, we were not able to express the mobile aspect of the "print" service. After trying to verify such a system we can say that the verification as a tool is inefficient because there are no adequate technologies in order to deal with mobile systems. We can, however, verify reachability or safety properties of the system.

Both approaches stress the lack of a temporal logic for higher-order process specification. An extension of CTL seems to be the correct approach. Some work exists on that subject [29], but higher-order processes are not taken into account and the mobility is only applied with names. Also, there is no tool and the proofs are built manually.

A second solution consists of translating a higher-order process specification into a first-order description and to establish properties on a logical model (built for the previous description). However, in this case, it is also necessary to extend CTL language to consider the exchange of names.

A suitable solution is to consider both: the verification of the system, as we have done above, and the validation of a prototype through simulations. In addition to proving key behavioral properties of our protocol, our formal method approach is also of value in creating a clear understanding of the structure of the agents, which can increase confidence in the correctness of a particular agent's system design.

In chapter 5, we will discuss different methods of implementing a mobile agent starting from a given formal language specification. The last step will be the development of a prototype of a mobile agent system starting from the specification of the HOπ-calculus.

Chapter 5

Synthesis Part

This chapter reveals the reflective process that we went through and which helped us to make a choice between the available mobile agent system technologies for the development of our prototype. We must remark that the temporal process also concerns the employed technologies, i.e., we used methods and approaches, which, at the time of development, were up-to-date. Nowadays, recent or improved versions are available and one reason for this is the fact that the mobility community is growing and developing quickly. We will focus on the feasibility of a prototype starting from its formal specification, which will allow us to simulate a mobile agent system. We will not address the problem of efficiency in this work. Three different ways and technologies to implement our prototype (through Remote Method Invocation (RMI) [50], Java Intelligent Network Interface (Jini) [4] and JavaBeans [76]), will be discussed here. The technologies RMI and JavaBeans will be used to implement the same case study, the SLP protocol, as described in the previous chapter. We have decided to choose a different case study to explain how the implementation with Jini works, as the SLP example will be treated in the next chapter, when an exact description of our prototype construction and implementation is given. We will also show that the unreliability of the first method leads to the employing of a second or third one.

This chapter is organized into three main parts: the first one confronts the reader with the introductory idea of the development and function of a mobile agent; the second one shows the three examples using different mobile agent technologies: RMI, EJB and Jini; and, finally, the last part, which is a discussion about the employed technologies and, consequently, a conclusion for further work.

We do not claim that our solution is the only viable one. The decision on which tech-

nology we choose was also based on our existing system requirements and on previous work.

5.1 Preliminaries Concepts

To develop an agent system, one needs a platform which allows distribution and mobility. One of the most common reasons to have distributed systems is to allow a set of agents to access shared resources or databases and to perform tasks depending on this data. This kind of system often involves building servers with functionality such as database connection pools, logging facilities or configuration tools. It is clear that some of these features are highly reusable and not limited solely to the original application. For example, mailing a message, communicating with a database and printing files are tasks that are commonplace in many kinds of applications.
For these reasons, developers often describe three kinds of code:

System logic is the kind of architecture code described previously; it can be reused in a wide variety of applications such as printing onto a specific plotter.

Business logic is code that is related to problems of specific domains such as intrusion detection systems or order processing.

Application logic is the part of the code that binds together various business logic parts to form a unified application that can be used by customers.

Typically, business logic is not bound to a particular application, but instead can be reused in many applications through the use of various kinds of application logic.
Because system logic is so common, we should be able to write code where the business and application logic parts are able to plug into a framework that provides system logic features, just as a desktop application would use an operating system to provide things such as a file system or environment variables. The object technology is well adapted to implement a system logic layer. More precisely, such an Application Programming Interface (API) already exists for the implementation of the system logic part. It provides guidelines for how to interact with the system framework. The Enterprise JavaBean (EJB) [76] specification is one such kind of API. It provides us with a standardized specification of how applications and business components can be plugged into any system logic implementation provider and how to have them executed by the framework in a clearly defined manner. Since EJB is a component-based development model one can build reusable components that can even be customized at deploy time without

touching the Java code. A component is a little more coarse-grained than just a Java class, so it is the next level of reuse. And since the components (i.e., beans) are deployed using an XML document that describes how the server should handle the bean, one can configure issues like security, transactions, resource use, etc. all declaratively, in XML, just by pushing a few buttons in a deployment tool. This file tells the EJB server which classes make up the bean implementation, the home interface and the remote interface. If there is more than one EJB in the package, it indicates also how the EJBs interact with one another. Most commercial EJB servers are supplied with graphical tools for constructing the deployment descriptor. JBoss [`http://www.jboss.com`] for example does have an XML editor, but it is just as easy to construct the deployment descriptor manually.
The system logic part is a suitable domain for an EJB approach. It includes many basic functions requiring technical values such as the domain name server, the mail server (both its protocol and its port), the names of all the printers on the network (with their IP addresses, the settings and their status), etc. An EJB is a kind of component, which encapsulates within a private part the technical aspects and provides us with a public part containing some meaningful service names. This information is network dependent and the system logic part must recognize these features itself without the help of the user.

The most important and widely used distributed object systems are the RMI [50] mechanism and the Common Object Request Broker Architecture (CORBA) [85]. RMI is a framework developed at the same time as Java, consequently compatible with the rest of our platform. We also chose RMI because it seems to be easier to master, it permits the administration of the access rights within Java and because we are building our system from scratch, with no hooks to legacy systems and fairly mainstream requirements in terms of performance and other language features. RMI is a distributed object system that enables one to easily develop distributed Java applications. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.
In systems like RMI and CORBA, the central connecting tissue between two programs are the client-side stub and the server-side skeleton. For a better understanding of how communication is accomplished in traditional remote procedure call systems, we will briefly explain the notion of stub and skeleton that we adopt from Jim Waldo, from Sun Microsystems in [101]. We nate that, with the RMI version 1.2, there is no skeleton use anymore.

5.1.1 Traditional Distributed Systems: Stub and Skeleton

The stub is a piece of code that implements an interface to a remote object or service in the address space of a client of that service. The job of the stub is to open up a communication channel to the server, convert all the arguments to be sent to the server into a form that can be transmitted across the wire, and dispatch those converted arguments. The stub code then waits for the response from the server, converting any return value from its wire representation to the internal form used in the process, and handing these results back to the program or thread that made the call.
The skeleton code provides similar functionality on the side of the server. The skeleton code receives the information transmitted by a stub, converts the information that has been transmitted over the network into a form that can be understood by the server program, and makes the appropriate up-call to that server program. That means that everything that can be transferred must be converted or encoded, and this represents a constraint because we cannot transfer an agent, for example, during its execution. The server program will return any result value to the stub code, which will translate these results into a form that can be transmitted over the wire, and send them back to the calling client (where the stub code receives them, as outlined above).
Comparing to the newest version of RMI where applying the "Proxy" pattern is sufficient, in the older version of RMI the code for the stub and the skeleton is produced by a compiler that takes as input, a definition of the interface between the client and the service written in some programming language-neutral declarative language (which, seemingly no matter what its form is, is called IDL). These compilers produce source code (sometimes in various languages) that can be compiled by the native compilers for the machines on which the client or service is going to run. The stub and skeleton can then be linked (either statically or at run time) into the client and the service.
A client, in such a system, uses a single stub to communicate with any service that implements a particular interface, and a service uses a single skeleton to talk to any client that is calling it through a particular interface. This works because the stubs and skeletons produced by the IDL compiler all correspond with a single wire protocol that is defined as part of the overall RPC system. This gives such systems their language- and processor-independence. It does not matter what the environment of the client is, nor the environment of the server. The system defines what is on the wire, and each system, both client and server, understands that protocol in a way that is appropriate to the language and environment running on the client or the server.
This independence from language and processor was a requirement when RPC systems were first invented. At that time, every computer company had their own processor, its own operating system, and one or more system languages. Furthermore, the computers

at that time were so slow that no one was willing to give up the efficiency of statically compiled binary code to obtain communication between machines. Since the computing world was unable to reach any consensus on a computing environment, communication was enabled by reaching consensus on the communication protocols between those environments.

However, systems based on protocols also have their limitations. Since the protocol needs to be translated into any possible language, the types of information that can be represented in the protocol must be limited to the kinds of data found in all of those languages. If some kind of information needs to be transmitted that is not a part of one of the target languages, conventions for that particular language must be defined to allow artificial representation in the environment, which always adds complexity and often subtly alters the information transmitted.

Systems that are based on a protocol must also fit the needs of all clients and services to the single protocol. The protocol must be completely general, and as is often the case when something needs to be good for everything, these protocols are often less than optimal for particular, specialized communications. RMI preserves the typing and the notion of an agent and they an be exchanged without being forced to encode or decode them manually. Therefore, RMI looks more attractive for us then TCP sockets, where everything must be encoded in binary (for example by MadKit project [51]). Protocol design, like any engineering design, is often a trade-off between efficiency and generality. In systems that are designed around a one-size-fits-all protocol, such decisions need to favor generality.

The most serious problems with such systems, however, is their rigidity once deployed. Since the skeleton code is part of the server, and the skeleton code is part of the client, any change in one has to be reflected in a change in the other. This means that if the server wishes to update its skeleton code, either to change the basic protocol, or to change the information transmitted to the server or returned to the client, the skeleton code in all of the clients using that server needs to be updated simultaneously. When client/server networks consisted of dozens of machines and the changes in services were slow, such simultaneous updates were possible. However, in the current network environment now grown to global proportions, where services are being defined and refined all the time, the requirement for simultaneous update has become a serious problem for protocol-based systems.

5.2 From HOπ Specification to RMI Implementation

Parts of this work were already published in [c]. We show the implementation using the SLP protocol described before. In order to have mobility within RMI, it is necessary to adopt an architecture which can ensure the disposal of mobile agents. This architecture is composed of four different parts and will be used for the SLP prototype. The code that is developed from the formal description of the Service Location Protocol is a mobile agent system prototype involving:

- the agents themselves
- an application server to run them on
- a client that creates and deploys agents
- and a management application that can work with agents running inside an agent server

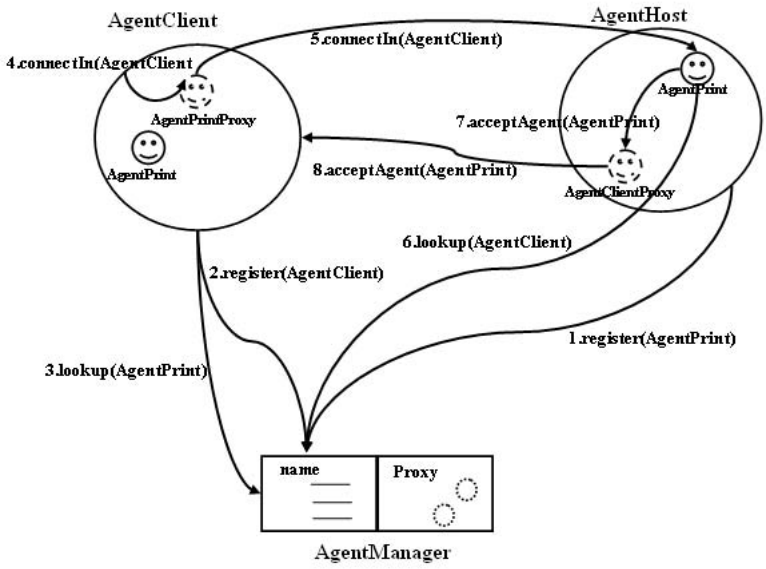


Figure 5.1: Our System

The key feature in the RMI framework of figure 5.1 is the dynamic class loading that allows us to have mobile objects whose code is not preinstalled on the servers they visit. We chose a particular way to implement the mobile aspects. This choice can be first explained by the definition of design patterns, which drive the translation from the

specification to implementation. These patterns are influenced by RMI. Creating a mobile agent system with RMI is quite straightforward and only requires some ability with network features. There are also four parts of code which play together on the network:

Agent host: the agent host corresponds to the SA (Service Agent) from section 4.1. Hosts are the equivalent of users and are used to host mobile agents. It is possible for a host to accept mobile agents, which are then made accessible as remote objects, so that someone can talk to the agent later on. It is also possible to list the currently hosted agents and remove agents when they are done with their tasks. Preferably, there should be a way for hosted agents to access resources at the host (for instance via security properties); otherwise there would be little reason for having them there.

Mobile agent: it corresponds to the service "Print" from section 4.1. It is important to implement a mobile agent as simply as possible. That is why we consider an agent as a remote object. The definition of mobile agents does not require the possibility to talk to them after their deployment. However, we added this feature by making them remote objects. For this reason, an agent consists of at least two parts: the remote interface that can be used to talk to it and the implementation of the remote interface that is the agent itself. After that, it can be instantiated and sent to an agent host if demanded, after which it will be exported as a regular remote object. The remote objects can then access the resources of the agent host in order to perform their perspective imported or local tasks (for our example to print a message, if the permissions are set up). In our case study, $AgentPrint$ can be considered as a mobile agent.

Agent client: the client that corresponds to the UA (User Agent), in our scenario, is responsible for creating an agent and sending it to the first agent host. The agent host accesses the implementation classes of the mobile agent through dynamic classloading. So, for this purpose, it is necessary to let the client run on a smaller web server (or "Agent provider") that provides the agent classes. Our choice of using a web server is due to the administration of the rights. It is, of course, possible to have only one web server in the system to serve this purpose, but in this way, the clients are completely independent of each other. This is why each client has its own web server.

Agent manager: it corresponds to the DA (Directory Agent) also from section 4.1. Although it is essential to control the entire system, we decided to add this component, which is used to manage and to interact with the agents that are hosted

on a particular server. It is able to ask the host for a list of its agents. It can then ask each agent for the management interface used for communication. The manager will not have any of the agent classes available when it starts, so it will have to be accessed through dynamic classloading. The agent manager is useful for observing the steps of the distributed application. For instance, the mobility of an agent is detected by the agent manager and its ability to view the state of all the agent hosts.

A component diagram is shown Figure 5.2, where each component or functionality is highlighted. A node of our system such as *AgentHost* can be described by several components. There is also an RMI registry that can be embedded in the Agent Manager

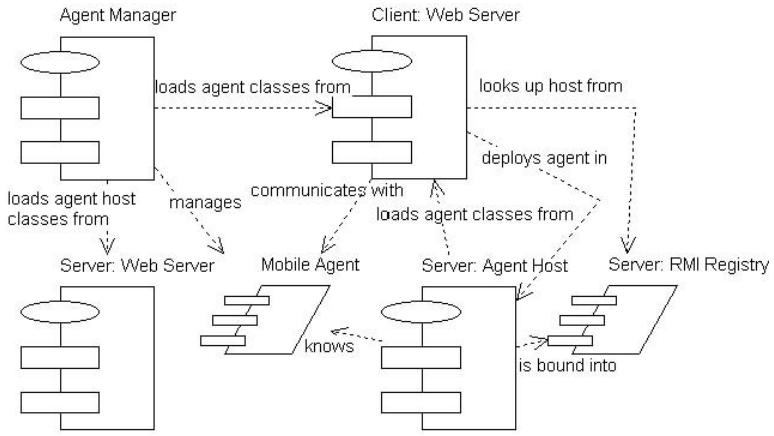


Figure 5.2: Design of Our System

that finds the agent host so that clients and managers can access it. Also, instead of having a web server in the server to allow the clients and any external registry to load the host's stub through dynamic classloading, it is possible to have an additional web server located at the client. Because of that, when the agent is sent to the agent host, the host can get the classes from the client's web server. So, in this scenario, we are using dynamic classloading in both ways. The web server located at the client would be used in order to download information like stubs or complete ".jar packages" .
Therefore, the presence of several web servers can be explained in 2 ways:

- There are physically several machines, therefore, for security reasons, we have at least one web server per machine.
- Because there are several directories, where it is possible to download data, we can add several web servers (e.g. one per directory).

In addition, when the agent manager accesses the agent host and gets the list of agents, it requires the classes for the stubs. Although these classes are not available from the agent host, the agent manager is able to get these from the client. This is done when the agent host receives the agent. It does not have the agent's implementation classes. It has to retrieve them from the client by creating a URL ClassLoader that points to the URL that is embedded in the remote call to the agent host. This class loader then loads the agent classes that have been associated with this class loader. This means that it is possible to know if an agent is local to a host or not. So, when the agent's stub is later sent to another client through a RMI call, the codebase annotation comes with it. Because the classes were loaded with a URL class loader, the URLs for this class loader are sent along with the call. The RMI implements the agent manager and client and then uses this information to access the classes from the client's web server. The notion of ClassLoader fits perfectly with the notion of exported data as described by R. Milner in the definition of the π-calculus. Each site who possesses its own class loader, also possesses a naming space. The partition is, thus, made between the original data and the copy on the receiver.
By default, an RMI program does not have a security manager installed, and no restriction is placed on remotely loaded objects. The *java.rmi* package provides a default security manager implementation that one can install with the following code:

```
if(System.getSecurityManager() ==null)
    {
        System.setSecurityManager(new RMISecurityManager());
    }
```

Java looks for a system-wide policy file in *java.home/lib/security/java.policy*, where *java.home* is the directory where the JDK or JRE is installed. If a security policy file is not specified, the JVM also looks for a user-defined policy file in *user.home/.java.policy*, where *user.home* is a user's home directory. The policy file syntax is described in the *docs/guide/security/PolicyFiles.html* file that is included with the JDK documentation. A sample policy file that grants full access permissions to everyone looks like:

```
grant
{
  permission java.security.AllPermission;
};
```

Policy files are used to grant permissions, represented by the Permission classes in the *java.security* package, to sets of classes or access grants to specific resources.

It is essential to note that the mobile feature in our architecture is not a move of an entity from one node to another, but, instead, a substitution (or rather a cloning). This concept is similar to the mobile feature present in the higher-order π-calculus language. A mobile object is sent as a parameter of a remote method on the host. When a remote object through RMI is exported to a client, it will be substituted with its remote stub. This concept is explained in Java semantics with the idea that remote objects that are not exported will not be substituted with their stubs, but will be sent as is through normal serialization. To sum up, the scenario is:

- to keep the remote objects unexported,
- to send them to the host,
- to export them there,
- to send them back (This corresponds to an exportation. The remote stub is returned to the agent, which is now at the server. The client can then use this stub to communicate with the agent. This is the core idea that our example is based upon.)

The agent interface represents the "contract" class of the mobile agent system. It is based on two remote interfaces and an exception class. The first remote interface is the one that is used by the agent's hosts, and the second is a remote interface that contains methods that all the agents must support. The exception is used by one of the host methods to indicate that the requested agent is not there. Figure 5.3 below shows the architecture of a mobile agent. In this special case the mobile agent needs another object in order to be sent or received. This object is not autonomous during its life cycle. Because this interface already extends "java.rmi.Remote", there is no need

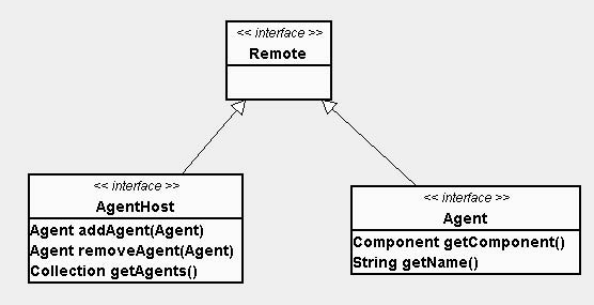


Figure 5.3: The Architecture of a Mobile Agent

for a sub-interface to do this as well. We need only to extend this interface. The *"getComponent"* method is used to get a Graphical User Interface component, which can be used to interact with the agent. The implementation of this can simply return some visual component that contains static information, and it is used to communicate with the agent. It needs to hold on to the stub of the agent so that it can call it. The last method is used to get a name (in Source Code 9) for the agent that is displayed.

Source Code 9 — *The Mobile Agent Interface in Java.*

```
1:  package mobility.agent.interfaces;
2:  import java.rmi.Remote;
3:  import java.rmi.RemoteException;
4:  public interface Agent extends Remote
5:  {
6:      public java.awt.Component getComponent()
7:          throws RemoteException;
8:      public String getName()
9:          throws RemoteException;
10: }
```

The server is divided into two main parts: the implementation of the remote interface that provides the actual service to be provided and a manager class that sets it up along with the necessary environment.

The agent host (Figure 5.4) is a remote object that implements the *"AgentHost"* remote interface. Its purpose is to keep a list of agents and to provide them with services they can use to perform their task. The construction of this component is based on the *"BeanContext"* API, which is a standard API for the development of JavaBeans. The agent is added in such a tree, and any service can be added similarly to make them accessible to the agents. In the manager class we will use an export method of the *"java.rmi.UnicastRemoteObject"* class to allow the client to receive the agent. The method *"to add the agent"* receives the agent as a regular object because it has not been exported to the client. Also, the first important thing to do is to export it. After this point, if the agent is sent as a parameter or a return value it will be substituted with the corresponding stub. Then, the agent is added to the host. Because the host class extends *"java.beans.beancontext.BeanContextServicesSupport"*, which itself implements *"java.util.Collection"*, the addition is handled by the *"BeanContextServicesSup-*

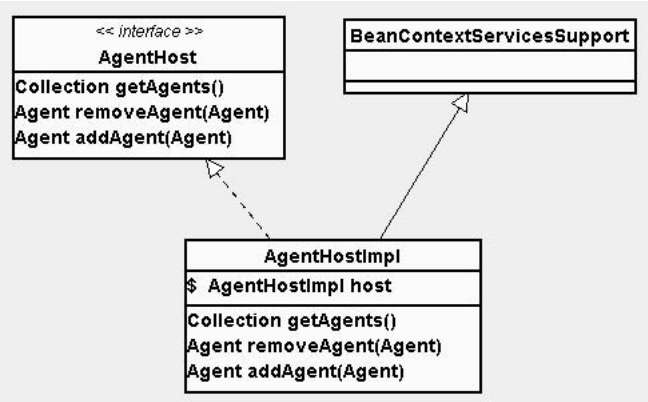


Figure 5.4: The architecture of the Agent Host

port" super-class. When the agent is returned, the client actually gets its stub because the agent is now exported. The client can then use this stub to call the agent remotely, just as with any other remote object. If a client wants to remove an agent for some reason, it can send the agent's stub to the *"removeAgent"* method. If the agent is found in the list it is unexported; thus making it yet a normal Java object in terms of how RMI treats it. So the agent is returned with its state. This state has to be serialized. The client can then extract any state it wants from the agent by calling it locally. Whenever an agent is requested several times, a cloning of the agent and its state will be carried out.

The manager class of the agent is shown in Source Code 10. The service startup code instantiates an agent host, exports it and binds it into the naming service. In this case, this service is called *"rmiregistry"*.

Source Code 10 — *The Manager Class of the Agent in Java.*

```
1:  public Main()
2:  {
3:      // Start server
4:      try
5:     {
6:          startWebServer();
7:          startNamingServer();
8:          startAgentServer();
9:          System.out.println("Server has been started and registered");
```

```
10:      }    catch (Exception e)
11:      {
12:          System.err.println("The server could not be started");
13:          e.printStackTrace(System.err);
14:      }
15: }
```

The implementation of the mobile agent begins by a definition of what it should do. Basically, our agent is an adaptation of the print service we described in the specification part 4.1. When the User Agent, which is the Agent client, asks its Directory Agent (the Agent manager), it wants to receive something that allows it to print. In addition, the agent will keep track of how many times it has been called and it will start up a thread that continuously prints the current state. Its state will be the visualization of the status print service. Another requirement for an agent is the ability to get a graphical user interface from it with which the user can interact. This part is not specified in our specification part, but it is a natural approach of a new service. The interface

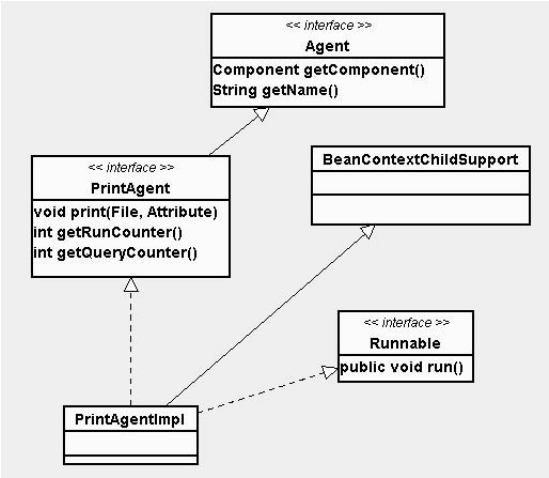


Figure 5.5: The Architecture of the Print Agent

"PrintAgent" (Figure 5.5) is a regular remote interface with one difference: It extends the agent interface instead. This means that the *"PrintAgent"* implementation not only has to provide the methods in the *"PrintAgent"* interface, but also the methods of the agent interface.
The class *"PrintAgentImpl"* implements the agent functionality. This agent extends

the *"BeanContextChildSupport"* class. This allows the agent to be placed within a surrounding *"BeanContext"*, which can be used to access other agents and services. Next it implements *"PrintAgent"* and the *"Runnable"* interface. Thus, it can start a thread that continuously prints its activity. The agent itself provides the code to be executed in that thread, which means that the *"Runnable"* interface is essential.
The attributes of an agent include:

- the name of the local host, which is used for the status,
- a runner thread, which executes the *"Runnable"* feature of this agent. This runner will belong to a *"ThreadGroup"*, which represents the set of all the mobile agents present and active on the host.
- a counter is used to keep track of the numbers of thread loops and to print a file with respect to the attributes.

The "print" method of the class *"PrintAgentImpl"* can be used by the caller to get an output on a printer in the room called "office". The counter is increased and the new status is printed on the screen.
The status code can be returned to the caller, but an essential feature is to use the critical section. It is used to prevent two or more clients from calling this object at the same time, which could lead to the counter being improperly updated.
The agent (Figure 5.6) in our scenario is a Service Agent and performs two things. First of all, it creates a Print agent, which is registered with the Agent manager. Secondly, it allows the user to call the print method on the agent. The client is composed of a

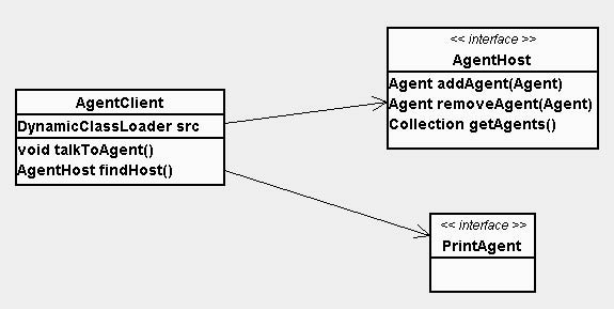


Figure 5.6: The Architecture of the Agent Client

web server, which is used to serve up the classes for the agent. The *"findHost"* method locates the host and then the agent can be created and registered with that host. In return, the user gets the stub to the agent so that the user can talk to it.

The Java-RMI solution we propose shows the capacity of encoding a specification. This solution is polyvalent, as parts of the code can be reused. Nevertheless, there are some limitations in using this approach, for instance "how to choose the right service when several services with the same signature are being proposed".

5.3 A HOπ Specification for a Mobile Agent in Jini 1.2

During the evolution of protocol-based distributed systems, the environment requiring such systems has changed dramatically. The safety of the Java technology environment provides a single, uniform environment in which code can be dynamically loaded into a running process no matter what the underlying processor or operating system is. Java's safety means that users are willing to allow such downloaded code to run. The result is a system in which portable binary code is available to the developer of distributed systems. It is this functionality that RMI, and through the semantics of RMI, Jini, exploit to change the protocol-centric nature of distributed systems.
In RMI and Jini version 1.2, the stub code used by the client is not owned by the client. Instead, that stub code comes from the service that the client is wanting to use. In RMI, such stubs are the RMI references, which implement not just the remote interfaces expected by the client but all of the remote interfaces supported by the service. In Jini version 1.2, these stubs are the proxy objects obtained from the Jini Lookup Service, which are often RMI references but need not be.
In both cases, the Java language equivalent of a stub is dynamically downloaded (if needed) to the client. The code that gets downloaded for the class comes from the service itself. Different services that implement the same interface may well have different code for their reference or proxy; a fact that is hidden from the client of the service who only needs to know the Java interface. But like any other object in a true object-oriented system, the implementation behind an interface can change without the client of that interface knowing or needing to know.
An immediate outcome of this is that the protocol between the client and the server is not the nexus of interaction between those two entities. Since the server provides the stub code to the client on the fly and on demand, that code can change. In particular, an RMI-based server can implement an extension of a previously supported interface without their needing to be any change on the part of the clients. The extension means there is new stub code for the client to use, but the client will receive that new stub code the next time it receives a reference to the service. This allows the service to change, and the clients to automatically update themselves on an as-needed (rather than on a coordinated) basis.

This also means that the RMI protocol (sometimes called JRMP) is an accidental, rather than an essential, feature of RMI. The protocol can be expanded and altered (within certain limits, caused by the RMI system's provision for distributed garbage collection) without changing the RMI system. Again, such changes are possible because the stub code, encapsulated in the RMI reference to the service, is a dynamically loaded and executed piece of code rather than something that has been associated (forever) with the client.

Jini and Protocol Independence

Jini proxy code is even less tied to a protocol than RMI. A Jini proxy object is only required to be an implementation of an interface (which is used to identify the object in the Jini Lookup service). This allows the client of the service to know what calls to make to gain access to the service. But how (or even if) the proxy communicates with the service itself is completely up to the proxy and the service from which it comes. The proxy can be an RMI reference (a common, and easily supported case), an object that communicates using some other common and well known protocol (such as CORBA's IIOP protocol), a piece of code that uses a specialized protocol known only to the proxy and the service itself, or a full implementation of the service that runs locally in the client's address space. All of this is transparent to the client, who only sees the Java interface. In the Jini world, the protocol used between a proxy and a service is a private matter between those two objects.

This works because the Jini version 1.2 system uses the RMI semantics notion of associating the proxy with the service and dynamically loading the proxy on demand. Effectively, the proxy and the service form a single object that is itself distributed, with part of the object living in the address space of the client and part of the object living at the location of the service.

This approach gives greater flexibility to what protocol is actually used. Different services can invent their own specialized protocols that are optimized for that particular pair of proxy and service. Protocols can evolve over time as new ideas are tried out.

But this also explains why the client's access to the network is Java environment-centric. For this approach to be viable, there needs to be a way of dynamically downloading code from the service to the client-code that the client can safely load into its address space and call. Java technology provides this kind of environment, and is the environment of choice for both RMI and Jini. The requirement of a service that wants to participate in Jini is that the service (be it hardware or software) be able to register a Java object in the Jini lookup service (and keep that registration alive throughout the renewal of

leases) that can communicate with the service. This Java proxy can use any protocol it likes to talk to the service. And the service can have the registration with the lookup service initiated and maintained through other Jini services, like those that are currently available in the Jini 2.0 beta release. Clients wishing to use these services need to be able to load and run the Java code that is the reference or proxy to the service.

5.3.1 Jini as Our Choice for the Implementation of an Agent System

Jini [4] version 1.2 as a tool based on Java, it enables us to realize, in a rather simple way, shared applications. It promotes the concept of a federation of services that collaborate dynamically over a network. A service, for Jini, is essentially a Java interface. Therefore, any object could be turned into a service. Some examples of services are printing, controlling a remote camera or an electronic fund transfer. Jini differs from other distributed architectures, such as RMI or CORBA, because it emphasizes a very dynamic approach. With Jini, as new services are plugged on the network, they immediately become available. For example, when one plugs a Jini printer on the network, it is immediately possible to use it from other Jini-enabled devices such as a PC, a PDA, or even a camera. For a few years, Jini [4] has been more and more essential on the framework market, allowing the development in distributed networks. Just as robots automate many aspects of manufacturing a computer, Jini automates and abstracts distributed applications' underlying details. These details include the low-level functionality (socket communication, synchronization) necessary to implement high-level abstractions (such as the service registration, discovery, lease and use) that Jini provides.
The ideas behind Jini corresponding to Jim Waldo's description (from Sun Microsystems) are simple: "Jini is designed so that chunks of code can find other chunks of code without people." Jini was designed with the assumption that the network is not reliable. Things join the network and leave the network. There is no central control. Also, Jini blurs the distinction between hardware and software, dealing only with services. In this case Jini represents the ideal framework for the type of mobility we are looking for: the mobile agents (as seen in section 2.2.3).
Jini architecture is based upon three distinct communication pathways, each with its own unique on-the-wire protocol (see Figure 5.7).

- Jini device services use a discovery/join protocol to announce their presence to, and register their access objects with, the Jini Lookup Service (JLS).
- The client in Jini version 1.2 uses RMI to communicate with the JLS, to enable it to retrieve these access objects by "type" (for example, PRINTER).

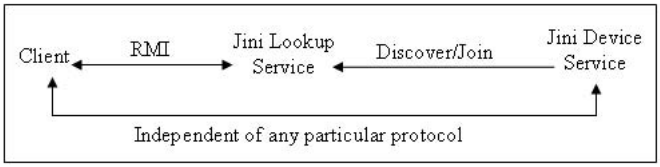


Figure 5.7: Typical Jini Configuration Protocols

- The client then uses the access object to communicate with the Jini service. This communication is based upon a "prearranged" protocol (for example, RMI, UDP, HTTP, IIOP) between the access object and the service from which it first originated.

As a result, when accessing the service, the client sees only the interface to the access object, and remains independent of the underlying protocol used to support the communication.
The objective of this section is to describe the modeling of mobile agents or code in Java integrated within the Jini platform. For a better understanding of their behavior and to validate their properties, such as for example, their return on their starting machine, it is necessary to formalize these aspects as we showed in the previous chapter.
The specification of our system will be given using the formal language higher-order π-calculus. In our case study, we are going to focus on the mobile aspect of agents that can move around between different servers to accomplish a task given by a task manager (`EMPLOYER`).

5.3.2 Case study using Jini 1.2

Our objective is to simulate a `MOBILEAGENT` that wants to get a job from a service collector called `EMPLOYER`. The `MOBILEAGENT` operates in a `ROUTE` system composed by a finite number of agent hosts, which we call `OFFICES`. Each `OFFICE` is able to make jobs and routes available by registering these to the `EMPLOYER` system. The `EMPLOYER` contacts the `MOBILEAGENT` to give him a certain job to do (in our case it will be *collect*) and a route map called `Mapchannel`. With this information, the `MOBILEAGENT` contacts the `OFFICE` directly, moves and gets the desired information from him (import of SQL orders into its bag). A `MOBILEAGENT` can also ask for services and contact many `EMPLOYERS`. It is possible to have more than one `EMPLOYER` service for a big `ROUTE` system. We can specialize many `EMPLOYERS` in order to make the offers of services manageable. To

simplify matters we will not discuss this option here. The `MONITOR` is a waiting room, where all the `MOBILEAGENTS` wait for some jobs. The `MONITOR` is also the starting and the ending machine for the `MOBILEAGENTS`. After carrying out the job, an acknowledgement *ack* will be sent back to the monitor. The higher-order aspect in our case study is represented through the `MOBILEAGENT` that contains the job to do (*collect*), the route map (*Mapchannel*) or even another agents. Figure 5.8 depicts this simulation. In

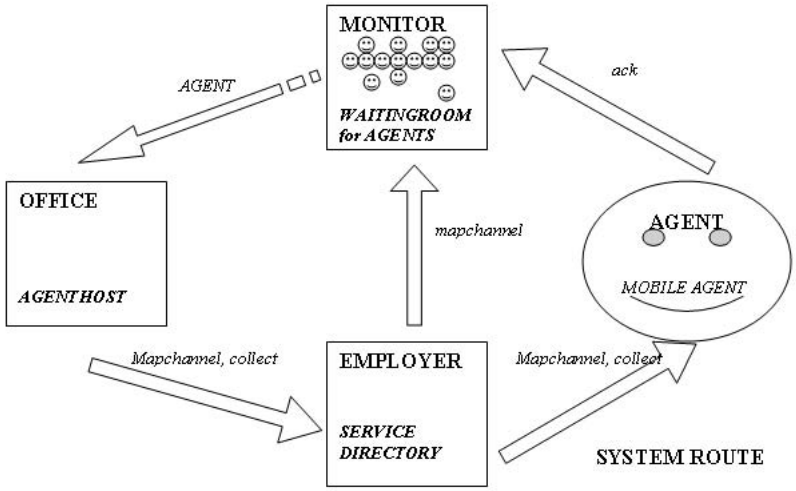


Figure 5.8: Our Route System

the context of our mobile agent framework, the agent host(s) provides Jini "collect" services. The mobile agent(s) is the Jini client. It can have a lease for a particular node and this can be modified during the action of the agent. Our "collect" service provides a set of SQL statements, which have to be executed later on a database. Jini services register with one or more Jini lookup-services by providing a service proxy for prospective clients. In turn, clients query the lookup service(s) for particular services. More detailed information about Jini structure are given by [4].

5.3.3 Specification Part

The higher-order π-calculus specification of our case study is given by our system $ROUTE$ and all elements are in a parallel relation to each other in order to be able to communicate.

ROUTE $= \nu\,(out, out_{AH}, out_A, ack, service)$

$OFFICE(out) \,|\, EMPLOYER(out, out_{AH}, out_A) \,|\, MONITOR(out_{AH}, ack)$

OFFICE(out) $= \nu(mapchannel, collect)$

$\overline{out}(mapchannel, collect)\,.\,mapchannel(agent)\,.\,(agent(o, s, a) \,|\, OFFICE(out))$

EMPLOYER(out, out$_{\mathbf{AH}}$, out$_{\mathbf{A}}$) $=$

$out(channel, service)\,.\,\overline{out_{AH}}(channel)\,.\,\overline{out_A}(channel, service)\,.\,EMPLOYER(out, out_{AH}, out_A)$

MONITOR(out$_{\mathbf{AH}}$, ack) $= \nu(MOBILEAGENT)$

$out_{AH}(x)\,.\,\overline{x}(MOBILEAGENT)\,.\,ack\,.\,MONITOR(out_{AH}, ack)$

MOBILEAGENT(out$_A$, service, ack) $=$

$out_A(channel, service)\,.\,\overline{ack}\,.\,MOBILEAGENT(out_A, service, ack)$

We specified as follows:

1. `ROUTE`: describes our entire system with all its components.
2. `OFFICE`: represents service-hosts, which make different services (jobs) available and notify their availability to the `EMPLOYER` service.
3. `MOBILEAGENT`: represents our mobile agents that are able to migrate to an `OFFICE` in order to apply for a job. The job is: *collect* database information and add it to its "bag".
4. `EMPLOYER:` plays a kind of reference book of all the tasks or jobs which are available on the local network.
5. `MONITOR`: is the waiting platform for the mobile agents.

We use gates like out, out_A, out_{AH} to specify the communication channel between the agents, employers, offices and monitor. In this specification the mobility is described by the channel parameters.

5.3.4 Implementation Part

Let us now take a look at a mobile agent framework that consists of two main components. This framework uses the Jini(1.2)/RMI platform given by [e] and implemented by [20]. The first component is the mobile agent `MOBILEAGENT`, that is an entity given some job to perform. The second component is the mobile agent host `MONITOR`, the service that provides the mobile agents execution platform. In a distributed environment, we can have one-to-many agent hosts as well as one-to-many agents. To be an active agent platform, a given node in the system must have at least one active agent host.
These two components map quite nicely to the Jini model. Jini, at the highest level, provides the infrastructure that enables clients to discover and use various services. Jini also provides a programming model for developers of Jini clients and services. In the context of this mobile agent framework, the agent host(s) provide(s) Jini services. The mobile agent (the `MOBILEAGENT`), is the Jini client. The Jini service, `OFFICE`, registers one or more Jini services by providing a service proxy for prospective clients. In turn, clients query the lookup service(s) for particular services that might be of interest. The Jini service *collect* will register at one or more Jini Lookups (in our case at the Jini Lookup Service 1, `EMPLOYER`). The analysis of the specification makes it possible to extract the emissions from higher-order, such as $\overline{x}(MOBILEAGENT)$. This exchange identifies:

- the support of the exchange: the channel "x",
- the mobile agent: `MOBILEAGENT`,
- the transmitter: `MONITOR`,
- the discoverer: `EMPLOYER`,
- the receiver: `OFFICE`.

From this point, we have the possibility of the application of our pattern $AgentMobile$ which consists to:

- develop the class `MOBILEAGENT` implementing the services imposed by the interface *"AgentInterface"*;
- develop the class `EMPLOYER` implementing the services implemented by the interface $AgentHostRemoteInterface$, $DiscoveryListener$ of the package $net.jini.discovery$.

- develop the class `MOBILEAGENT` publishing the object of the class *collect* to the directory Jini (reggie) via a fixed service, $JoinManager$ of package *"package.net.jini.lookup"*. Reggie can be replaced by a web server.

Agent host construction

In our case study we will represent the agent host through `MONITOR`. The `OFFICE` will publish a service at the corresponding Jini Lookup service.
The first step in building the agent host is to create a remote interface, the service template that agents will look for via the Jini lookup service. The *AgentHost Remote* $Interface$ provides one method, $acceptAgent()$, which agents call to travel to the implementing agent host as in Source Code 11.

Source Code 11 — *AgentHostRemoteInterface in Java.*

```
1:  public interface AgentHostRemoteInterface extends Remote {
2:      public void acceptAgent (AgentInterface ai) throws
    RemoteException;
3:  }
```

The advantage of Jini is that objects can publish several interfaces that provide multiple services. For instance, if we had a distributed datawarehouse, we might have an agent host that provides a local data access service. In this instance, a data-mining agent might look for a host that provides the data access service and move to that host to perform localized mining operations. Therefore, we can have agents with different missions share hosts that provide multiple services.
The second step in building the agent host is to provide an implementation of this remote interface that is the actual Jini service *collect*. We have provided a $MobileAgentHost$ class that implements the $AgentHostRemoteInterface$.
Figure 5.9 shows the class diagram for $MobileAgentHost$. The class extends the *"java.rmi.server"* package's $UnicastRemoteObject$ class, which allows clients to obtain a remote reference and call its methods. The $MobileAgentHost$ also implements the $ServiceIDListener$ interface, which is passed as a unique $ServiceID$ object via the $serviceIDNotify()$ method when the service first registers with a Jini lookup service (1 and 2).
The $MobileAgentHost$ constructor takes the *agentObject* object reference stored as

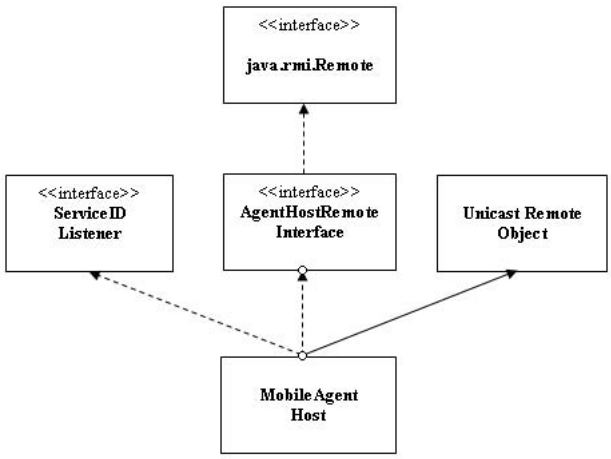


Figure 5.9: Mobile Agent Host Class Diagram

member data and passed to arriving agents via the $doWork()$ method. The constructor itself performs two basic functions. First, it creates a *LookupDiscovery Manager* to locate the Jini lookup service. Second, it creates a $JoinManager$, with the $LookupDiscoveryManager$ as a parameter, to add this lookup service (for example *collect*) to the Jini service federation. In the $acceptAgent()$ method implementation, the $MobileAgentHost$ binds incoming agents to an $AgentThread$ as in Source Code 12.

Source Code 12 — *acceptAgent Method in Java.*

```
1:  public void acceptAgent(AgentInterface ai) throws RemoteException
2:  {
3:      AgentThread at = new AgentThread(ai);
4:      at.start();
5:  }
```

In turn, an inner class instance, $AgentThread$, is created to run the bounded agent by calling its $doWork()$ method, passing the $LookupDiscoveryManager$ and agent object references. For each arriving agent a new thread is created (see Source Code 13):

Source Code 13 — *AgentThread*

```
1:  private class AgentThread extends Thread
2:    {
3:     private AgentInterface myAgent = null;
4:     AgentThread(AgentInterface ai)
5:      {
6:        myAgent = ai;
7:      }
8:     public void run()
9:      {
10:        myAgent.doWork(myLDM, myAgentObject);
11:     }
12:   }
```

Agent construction

In our case study the Jini client is the mobile agent `MOBILEAGENT` which collects the data *collect*. The `MOBILEAGENT` receives a proxy object from the Jini Lookup Service and it thus becomes able to communicate with the service through the proxy. The first step in building an agent is to create an interface for remote services. For this, we create an *AgentInterface* that extends the *Serializable* interface. The *Serializable* interface marks the implementer as a serializable entity, or one that can be sent across the wire.

Source Code 14 — *The Serializable Interface.*

```
1:  public interface AgentInterface extends Serializable
2:  {
3:      public void doWork(LookupDiscoveryManager ldm, Object workObject)
    ;
4:  }
```

The *AgentInterface* consists of a *doWork*() (Source Code 14) method that is called when an agent arrives on a given host. This method takes two parameters. The first parameter is a reference to the *LookupDiscoveryManager* maintained by the current

host. The agent uses this reference if and when it decides to look for new service providers, such as when it wants to travel to a new agent host. The second parameter is an optional (possibly null) object parameter, which contains data necessary for the agent to complete its job. The second step in agent construction is providing an $AgentInterface$ implementation for which an abstract $MobileAgent$ class was created. This class constructor builds a service template that locates services of type $MobileAgentHostInterface$. It also provides three additional methods: $doWork()$, $moveToRandomHost()$, and $getMobileAgentHosts()$:

1. To perform an agent-specific task, subclasses override the abstract $doWork()$ method.

2. When the agent wants to move, subclasses call the $moveToRandomHost()$ method, which performs the following three steps:

   - Get a list of the currently available mobile agent hosts with a call to $getMobileAgentHosts()$.
   - Randomly select a host from this list.
   - Move to a new host by calling the a$cceptAgent()$ method. If the call on the selected host fails, select a new host.

3. To obtain a list of currently available agent hosts, subclasses call the $getMobileAgentHosts()$. This process requires the following steps:

   - Call $getRegistrars()$ to obtain a current list of lookup services.
   - Iterate through each lookup service to find services that match the desired template; in this case, $AgentHostRemoteInterfaces$. Note that we might have duplicated the same agent host registered with multiple lookup services. The $myMOBILEAGENTServiceTemplate$ object, a $ServiceTemplate$ class instance, passed to the lookup() method initializes in the $MobileAgent$ constructor.
   - Add each matching service to a vector of $AgentHostRemoteInterfaces$.

Diagram 5.10 depicts the interactions between the $MobileAgent$, $MobileAgentHost$, and the Jini lookup service.

The solution we propose within this section uses mobility and, even if the result is more complicated, it is progressive, in particular because of the administration of lookups.

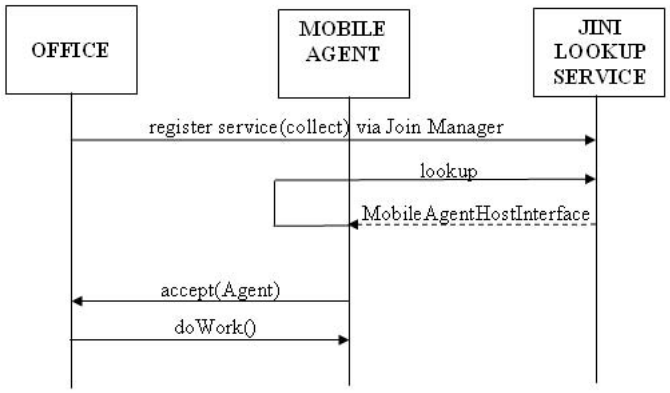


Figure 5.10: Diagram for interactions between MobileAgentHost, MobileAgent, and Jini Lookup service

5.4 Mobile Agent with an Enterprise JavaBean Approach

Traditionally, a protocol such as SLP [31](that we described in this thesis as the major case study) was used for the recognition of services for printing and mailing. This protocol is based on mobile aspects where the application logic part queries, for example, the location of a particular printer and, depending on where this part is executed, a directory agent provides one or more specific services. This protocol allows the client to download codes for particular network services as necessary. Thus, the application logic part can interact with the downloaded network service driver through a standardized API defined for the service.

We use an EJB structure to hide the technical aspects, which ensure mobility. With this approach, we offer a framework of Java components that are inherently mobile. We present the basis of our approach and its consequences in an n-tier application. First, we introduce the definition of the EJB component and its mobile version. Then, using as an example our case study SLP, we explain how it can be implemented. Finally, we compare our solution with previous works based on Jini technology and conclude with the domain dependent technology. Our interest is also the employment of application servers such as JBoss or Jonas [`http://jonas.objectweb.org`].

5.4.1 Mobile component approach

The EJB specification declares what is essential in order to enable a component to be plugged into an EJB container and also what the EJB container must do with the component. This specification defines an API that the bean provider can use to interact with the container and another API that the container can use to interact with the component. Because it is preferable to allow many kinds of implementations of system logic layer, most of the specification is declarative in the sense that it is essential to declare what we want to happen, but not how it should happen.
For example, a component can declare that it wants to access a particular printer, that it wants exchanges of data to be handled in a particular way or that only a specified set of users can call it. Exactly how the container decides to implement this request is irrelevant as only the result is important. This allows the EJB component to be fairly loosely coupled to any particular container and, as a consequence, it can be used with various containers without any code changes.
Once components have been deployed in an EJB-container, it is necessary to access them somehow. This is done by the use of Remote Method Invocation (RMI) [50]. So it is important to use a remote interface, stubs and remote exceptions.
A container should provide the following services:

Transaction Some actions in a component can involve several steps, and, especially if an automaton is used such as a database or a printer, where each page is sent to the printing service. A transaction contains all of these actions to perform. It represents a sequence of atomic actions that either completely succeed or completely fail. EJB allows the bean provider to declare which transactional requirements the components have. The EJB container implements the requirements by coordinating printer interaction accordingly. EJB containers use the Java Transaction API (JTA) and Java Transaction Service (JTS) to do this.

Security EJB allows the bean provider to declare who is allowed to call the operations of the components. The EJB container then authenticates the client and makes sure that they are only able to invoke methods that they are allowed to access. EJB uses the Java Authentication and Authorization Service (JAAS) to perform this. JAAS is an API that allows clients to declare their identity and servers to access this information for authorization purposes.

Environment properties The component may have features that are configurable. Because of this, EJB allows the bean provider to declare the environment properties and their respective values. The component can then look up the values of

their respective properties at runtime. This makes it possible to configure components without having to either hardcode values or to provide custom configuration files.

Distribution Because EJB is a framework for distributed components, there needs to be a way for clients to invoke the components through the JNDI naming system and allow clients to access them through RMI. EJB containers may use the RMI over Internet Inter-Orb protocol (IIOP) technology to allow Common Object Request Broker Architecture (CORBA) clients to access EJB components.

Server connectivity Because components can typically access any server, such as mail servers or database servers, EJB must be able to provide this functionality. Components can declare their server needs via an XML description;

Component relationships Components seek to make use of other components. For example, application logic components typically access business logic components in order to perform certain complex tasks. EJB allows the bean provider to declare which other components a specific component wants to access, and the EJB container then makes those components easily accessible.

EJB uses many other standard specifications such as JTA and JDBC to perform its action. Many of the preceding features are declared by letting the bean provider write a special XML file with descriptions of the component and its needs and requirements.

5.4.2 Printing Component Example (PrintEJB Component)

JBoss, as an EJB server has several interesting features with regard to RMI and Jini. Instead of making each component a remote object, there is one object to which all calls go. This means that only one remote stub has to be generated. This is done when the EJB container itself is compiled without requiring user interaction. Accessing the components without having to make remote objects is realized because the single remote object contains the following remote method:
public java.rmi.MarshalledObject invoke(java.rmi.MarshalledObject mo);
These marshalled objects contain information about which component the call is meant for, which specific component instance should get it and the method and the parameters for the actual call. With this information, the container can forward the call to the right instance for invocation.
For the construction of our case study, a manager of objects is not necessary as the EJB container implements the manager functionality for their users itself by taking care of

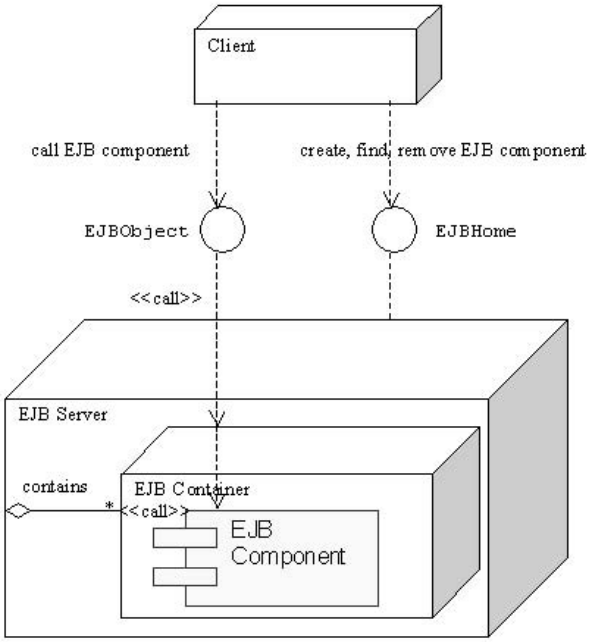


Figure 5.11: EJB container principle

the lifecycle of the object and placing references to it in the naming service. The user can, thus, solely focus on the behavior of the components and ignore the administration. By taking care of the routine tasks, the business logic developer can focus and spend more time on writing the business logic code instead of the system level code of the component. Now we will walk through the different parts of our components called *"PrintEJB"* and *"MailEJB"* and conclude with a client that uses them. The structure is similar for both kinds of components.

The PrintEJBHome Interface

The primary difference compared to an ordinary remote interface is that it extends *javax.ejb.EJBHome* instead of *java.rmi.Remote*, and it contains a couple of methods that are always available to the clients: Source Code 15.

Source Code 15 — *PrintEJBHome Interface in Java.*

```
1:  import java.rmi.*;
2:  public interface PrintEJBHome extends javax.ejb.EJBHome
```

```
3:  {
4:    public Print create() throws javax.ejb.CreateException,
    RemoteException;
5:  }
```

The method that has been declared is used to create *"EJBObject"* to talk to the stateful session bean. The *"EJBObject"* implements our interface "Print", which will be looked at next. The create method also throws the EJB-specific exception if the creation fails. As usual, this is a remote method that also throws the java.rmi.RemoteException. The EJB container provides the implementation of this object and makes it available through JNDI.

The PrintEJBRemote Interface

This is the interface (Source Code 16) used to communicate with the component once it has been created through the *"EJBHome"* object.

Source Code 16 — *PrintEJBRemote Interface in Java.*

```
1:  import java.rmi.*;
2:  import java.text.*;
3:  public interface Print extends javax.ejb.EJBObject
4:  {
5:       public void print(String fileName, Format f) throws
    RemoteException;
6:  }
```

As with *"EJBHome"*, this remote interface does not extend *java.rmi.Remote* directly, using instead the *javax.ejb.EJBHome* interface. And, just as with *"EJBHome"*, the *"EJBObject"* interface contains methods that are available regardless of what the component does. This EJBObject interface looks like a regular remote interface: the method parameters and return value can be serialized, and the throw clause must include the *java.rmi.RemoteException*. We can already conclude that this solution depends also on RMI.

The PrintEJB Bean Implementation

The component implementation (Source Code 17) is fairly straightforward. The biggest difference compared to regular Jini approach is that any EJB component must implement the proper sub-interface of *javax.ejb.EnterpriseBean*. In this case, as a session bean is being developed, the *javax.ejb.SessionBean* interface is implemented. This contains various callbacks that the container uses throughout the lifecycle of the component instance. For example, once it has instantiated the Print class, it calls *"setSessionContext"* to give the component an object that can be used to access the container.

Source Code 17 — *Component Implementation in Java.*

```
1:  import java.rmi.*;
2:  import java.text.*;
3:  import java.awt.print.*;
4:  public class PrintBean implements SessionBean, Print
5:  {
6:      String name;
7:      public void print(String fileName, Format f) throws
    RemoteException
8:      {
9:          FileOutputStream outstream;
10:         StreamPrintService psPrinter;
11:         String psMimeType = "application/postscript";
12:
13:         StreamPrintServiceFactory[] factories =
14:             PrinterJob.lookupStreamPrintServices(psMimeType);
15:         if (factories.length > 0)
16:         {
17:             try
18:             {
19:                 outstream = new File(fileName);
20:                 psPrinter =  factories[0].getPrintService(fos);
21:                 // psPrinter can now be set as the service on a
    PrinterJob
22:             }
```

```
23:                     catch (FileNotFoundException e)
24:                 {
25:                     e.printStackTrace();
26:                 }
27:             }
28:         }
29: }
```

The component implementing the *javax.ejb.SessionBean* interface also has an attribute and a name. Because EJB components do not have to implement the remote interface, it is easy to get them to mismatch. This will not lead to compile time errors, but will cause the deployment within the container to fail. To get around this, the remote interface is split into two parts: one regular interface with all of the methods and another that extends the first interface and the *javax.ejb.EJBObject*. The implementation of the EJB calls corresponding to the home interface is as follows in Source Code 18.

Source Code 18 — *EJB calls in Java.*

```
1:  public void ejbCreate() throws CreateException
2:{
3:    try
4:    {
5:      // Retrieve the name from the environment settings
6:      name = (String)new InitialContext().lookup("java:comp/env/print");
7:    } catch (NamingException e)
8:    {
9:      throw new CreateException("Could not get name for component");
10:   }
11:}
```

Although the *"EJBHome"* interface does not contain a method called *"ejbCreate"*, it does contain a *"create"* method. When the client (UserAgent) calls create, the container may create an instance and call *"ejbCreate"* on this new instance. The instance may then be used to execute calls to the *"EJBObject"*. However, because the instance is

not bound to the UserAgent, the container may use this instance to handle calls from another UserAgent as well. This mechanism is called instance pooling. Because of this, when another UserAgent calls the create method on the *"EJBHome"* object, the container may find that it has already created a component instance and will not create another.
An EJB component is not composed entirely of Java code. It also contains a descriptor as in Source Code 19, that tells the EJB container how to deploy and execute the component. The descriptor for our case study contains information about the component called $PrintEJB$, as well as security roles and permissions.

Source Code 19 — *Descriptor in XLS.*

```
<ejb-jar>
   <description>PrintEJB case study for the SLP example </description>
   <display-name>PrintEJB </display-name>
   <enterprise-beans>
    <session>
            <display-name>PrintEJB for SLP</display-name>
            <ejb-name>slp.printejb.interfaces.PrintEJB</ejb-name>
            <home>PrintEJBHome</home>
            <remote>slp.printejb.interfaces.PrintEJB </remote>
            <ejb-class>slp.printejb.interfaces.PrintBean</ejb-class>
            <session-type>Stateful</session-type>
            <transaction-type>Container</transaction-type>
        <session-env>
            <env-entry-name>name</env-entry-name>
            <env-entry-type>java.lang.String</env-entry-type>
            <env-entry-value>PrintEJB</env-entry-value>
        </session-env>
    </session>
   </enterprise-beans>
   <assembly-descriptor>
    <security-role>
        <role-name>Guest</role-name>
    </security-role>
    <method-permission>
        <description>Guest Access</description>
        <role-name>Guest</role-name>
        <method>
        <ejb-name>PrintEJB</ejb-name>
```

```
        <method-name>*</method-name>
        </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>
```

This first part contains a description of every component that is included. Next, the deployment descriptor that deals with how the container should execute this component is described.

A Client

The client for this component uses JNDI to look it up, and it is called in a similar way as with RMI; the main difference being that it is essential to create it before an *"EJBObject"* forms the *"EJBHome"* object. The code is given in Source Code 20. This approach corresponds to the REV paradigm from section 2.2.3.

Source Code 20 — *A client in Java.*

```
1: public class PrintClient
2: {
3:  public static void main(String[] args) throws Exception
4:    {
5:        new PrintClient();
6:    }
7:    public PrintClient() throws Exception
8:   {
9:    PrintEJBHome home;
10:   try
11:   {
12:     Context ctx = new InitialContext();
13:     home=(PrintEJBHome) PortableRemoteObject.narrow
14:         (ctx.lookup("PrintEJB"), PrintEJBHome.class);
15:     ctx.close();
16:   }
```

```
17:     catch(NamingException ne)
18:     {
19:         System.out.println("Could not lookup PrintEJBHome");
20:         throw ne;
21:     }
22:     Print printEJB;
23:     try
24:     {
25:       printEJB = home.create();
26:     }
27:     catch(CreateException ce)
28:     {
29:       System.out.println("Could not lookup PrintEJB");
30:       throw ce;
31:     }
32:     String filename = "report.txt";
33:     try
34:     {
35:       printEJB.print(filename, new Format());
36:     }
37:     catch(RemoteException re)
38:     {
39:       System.out.println("Could not get print from PrintEJB");
40:       throw re;
41:     }
42:     finally
43:     {
44:       printEJB.remove();
45:     }
46:   }
47:}
```

The *"EJBHome"* object is first looked up from the JNDI namespaces. The use of the class *"PortableRemoteObject"* is useful because the container may be CORBA-based,

in which case an externalized reference is stored in JNDI instead of a real object. The narrow() call then replaces the externalized reference with a real object implementing the given Home interface. The EJB server is responsible for starting the namespaces that clients can use to access the components. When the component is deployed, the server binds the created EJBHome object in the JNDI namespaces. Next, we create an *"EJBObject"* that implements the component's remote interface and we then invoke the print() method on it. The call is forwarded by the container's *"EJBObject"* to the component and sends the file to the printer. Using these methods we encounter some disadvantages, such as the large number of services that must be started, the use of a web server and the use of a container and a descriptor (with the system configuration) in order to ensure that the communication can be managed successfully.

5.5 Discussion and Conclusion

In the first example (5.2) we have shown a way to implement an agent system proceeding from a formal specification of the system, using the RMI method. The question now is whether or not the RMI is the only and/or best solution for agent implementations. It is obvious that RMI is not the only solution and, if we look at the newly-emerging Java technologies for building distributed computing systems, we find, among other technologies, the framework Jini and RMI provided by Sun. The main difference between the two concepts of RMI proxy and Jini proxy, is that the Jini proxy concept is protocol independent and the proxy carries out requests by itself or it can also use an RMI call or its own proxy provider. Also, using RMI in mobile systems, we cannot really move the code as in Jini, but only a copy of it. We do move a copy as well in Jini, but in this case it concerns a complete copy. When the service is published a total copy will be transferred to the Lookup service (in this case, if the service provider is stopped or killed, the service will survive on Lookup). Also a total copy will be moved at the time of requesting the service (the client receives a copy and that permits other agents to request the same service). These features are innovative and useful for a network, but they also lead to difficulties, which concern the service state or information during its execution. By using Jini we also obtain service-provider independence. Jini clients do not need to know where a service is located; they simply use the discovery mechanism to obtain a proxy to the service they want to use. Conversely, in RMI (Remote Method Invocation), one must know the URL of the server he wants to use. That logical next step convinced us to provide an implementation within the Jini platform.
The number of supporters for Jini are increasing every year. We are convinced that Jini is a good choice for the implementation of our prototype for an agent system , especially

given that it supports better small to medium-scale applications and offers loosely coupled federations with dynamic administration. After the implementation of the second example (5.3), we show another technology used for the development of mobile agents because of comparison matters.
Nowadays, many global applications with complex business logic and potentially thousands of concurrent users are developed on J2EE compliant platforms, because it supports large-scale systems and provides a centralized service.
The last example 5.4 gives an approach of our prototype using EJB.

EJB or Jini?

EJB versus Jini is a problem, which is no easier to solve using such a simple example. The feature set provided by Jini is quite similar to those of EJB. They both provide transactions, distributed services and lookup services. Many EJB containers also provide fault tolerance and scalability features, as do Jini's to some degree. So the following question may seem logical at first: Should we use Jini or EJB?
However, the question has a serious flaw. The problem is that the functional overlap is only superficial because the two specifications serve different purposes: Jini is primarily used to create system logic code, and EJB is used to create application and business logic code. Hence, the system logic code that is used to implement an EJB container can use Jini to provide its services to the EJB components that have been deployed in it. For example, the JNDI implementation of the EJB container can use the Jini lookup service as the basis of the implementation, and the failure features of the container can take advantage of Jini mechanisms to handle service failures.
For this reason, the question should really be: should we use EJB and Jini? And the answer could very well be yes, since they complement each other rather well. It is also quite possible to expose EJB components through the Jini lookup service; that is, use the Jini lookup service to wrap the JNDI implementation of the EJB container. This allows Jini clients to access EJB components as though they were a standard Jini service.
In our approach of a mobile protocol like SLP with the Enterprise JavaBeans technology, the Enterprise JavaBeans introduces an API and semantics for that API, which allow system logic to be written so that application and business logic components can easily be plugged into it. This allows the component developer to focus on the problem domain of the application and use the features provided by an EJB implementation provider to supply the system logic parts.
This skeleton seems to be a suitable scheme for the use of mobile features based on the

Jini framework. Yet neither framework is concurrent and both have different applicability. They have differing degrees of robustness depending on network traffic and it is not possible to ensure the same fault tolerance with both approaches. This is why EJB API and Jini API are neither interchangeable in mobile code nor transaction architecture. As a conclusion we can confirm that the Jini society is distinguishing itself more and more, and that Jini will become a more stable and viable solution.

Chapter 6

A Practical Approach

This chapter proposes a practical approach for a mobile system agent, making use of our previous results. We have implemented a prototype, called *HOPiTool*, which offers the possibility to validate mobile agent systems conceived with higher-order π-calculus and to automatically generate Jini code. This prototype can be enriched with a model checker in order to verify properties of the system.

6.1 Related Work

Models written according to the formal language HOπ can be simulated with our simulation tool *HOPiTool*. Additionally, code for them can be generated following a straightforward mapping to Java classes considering the use of a mobility support platform. Altogether we have a framework for the development of mobile code applications. The innovative aspect of the framework is the use of HOπ formal language as the underlying unifying formal method. Our achievements can be characterized by the mapping of a HOπ specification into a simulation model and the generation of code for a mobility support platform. Under these assumptions we could not find in the literature projects that address the same points that our environment does attempt to address. According to our review from section 2.1.3, we have found projects that use formal methods to develop applications based on code mobility; some of them providing analysis tools. The KLAIM [75] (Kernel Language for Agent Interaction and Mobility) project aims at the creation of a language supporting the paradigm of code mobility programming. KLAIM explicitly uses localities for accessing data or computational resources, and has a type

system to control access rights. In the KLAIM structure there is a net coordinator that has control over the net (there is a special coordination language) and processes. One special feature of KLAIM, that is similar to our project is the possibility to generate code through the Java implementation of KLAIM, named KLAVA. There are also projects available that use formal verification techniques. Although planned, our environment does not yet address this point. The main projects under this approach are MobiS [71], Mobile UNITY [35] and Mobility WorkBench [99].
MobiS is a specification language whose specifications denote a tree of nested spaces that dynamically evolves over time. There is the possibility of automatic verification of specified properties. Mobile UNITY, unlike MobiS, is a model to analyze key concepts and ideas of mobility. The Mobility WorkBench is a tool and not a specification language. It has been designed to analyze concurrent systems over specifications written in the polyadic π-calculus formalism. One major advantage of using a simulation model is the possibility of validating strategies as well as control algorithms for complex cases where the use of formal verification could lead to a state explosion problem. These models may be used to check if the components of an application behave as expected, if they are independent from each other such that the replacement of a simulated component by a more sophisticated version becomes possible and also to check the application behavior under various environment conditions. Of special concern for open systems, we have a tool and a method that allow us to interactively reason through simulation about mobile applications in the presence of selected failures, helping the developer to formally specify robust applications.

6.2 Brief Introduction to Jini 1.2

Making the computations mobile means that agent hosts have to explicitly decide where in the net a computation should take place. The term "mobile agent" captures best the view that computations can decide for themselves to move or to stay. There have been various systems proposed to implement mobile agents. The common feature is the mobility of code between the agent hosts. Each host becomes an agent execution environment (i.e., virtual machine or interpreter for portability reasons) that understands a common set of agent instructions or a common programming language. One of the newest platforms is Jini [78] from Sun. At the highest level, Jini provides an architecture enabling a distributed system for services to become available on a network. The Jini architecture exists in a layer above the Java technology, but below the application and services.
The infrastructure is a set of components that provides for federating services in a distributed system. It provides mechanisms for devices, services, and users to join and

detach from a network. Joining into and leaving a Jini system is an easy, often automatic, occurrence. The infrastructure of a Jini system includes the following:

- A distributed security system, integrated into RMI, which extends the Java platform's security model to the world of distributed systems.
- The discovery and join protocols; service protocols that allow services (both hardware and software) to discover, become part of, and advertise supplied services to the other members of the federation.
- The lookup service, which serves as a repository of services. Entries in the lookup service are objects that can be downloaded as part of a lookup operation and act as local proxies to the service that placed the code into the lookup service.

Jini technology uses a lookup service, with which devices and services register. When a device plugs in, it goes through an add-in protocol, called discovery and join-in. The device first locates the lookup service (discovery) and then uploads an object that implements all of its services interfaces (join). The lookup service acts as an intermediary to connect a client, looking for a service, with that service. Once the connection is made, the lookup service is not involved in any of the resulting interactions between that client and that service. Figure 6.1 depicts how this all works in the Jini architecture [79]. As

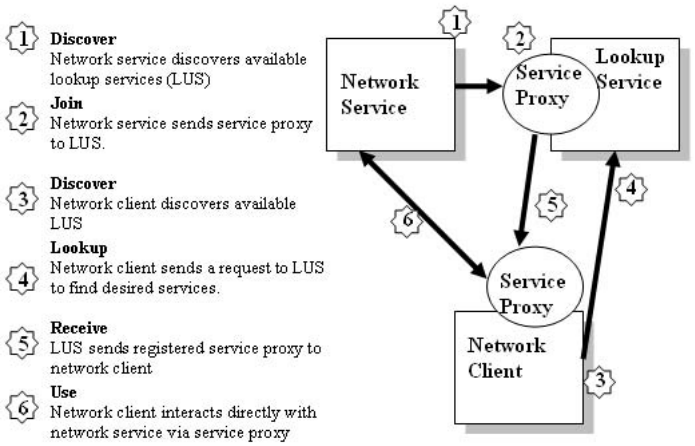


Figure 6.1: How Jini technology works

part of the system, the service must first register itself with the lookup service through RMI. Upon service registration the service's proxy is uploaded onto the lookup service. The client can initiate communication with the service device by requesting the service

from the lookup service and the service proxy. The communication with client and server can take any form defined by the provider of the service. The details of this can be found in [77].
Jini network technology consists of an infrastructure and programming model that addresses the fundamental issues of how clients and services discover and connect with each other to form an spontaneous community. Written entirely in Java language and utilizing its object-oriented features, Jini technology uses the mechanisms pioneered by the Java Remote Method Invocation API to move objects around the network.
Services employ a proxy (an object with service attributes and communication instructions) to move around the network. Through the processes of discovery and join, services are found and registered on a network. Registering means that the service has sent a service proxy to all lookup services on the network, or a selected subset. A lookup service is equivalent to a directory or index of available services, where proxies to these services and their code are stored. When a service is requested, its proxy is sent to the requesting client so that the service can be used. After that, the proxy conducts all communication between the client and the service.
To preserve the network flexibility and resilience, Jini technology introduces the concept of leasing. When a service joins the network, it registers its availability for a certain leased time. Before the time expires, the service may renegotiate the lease. If the service is removed from the network, its entry in the lookup is removed automatically when the lease expires.

6.3 The Architecture of our Prototype

The specification and validation platform architecture is where functionalities are directed. Figure 6.2 below emphasizes, not only the functionalities, but also the dependencies of data. The rectangles represent the functional entities, whereas the parallelograms are strategic data. Our *HOPiTool* prototype consists theoretically of five parts:

1. The graphical editor used to write a specification down.
   The objective is to manage a set of perspectives. Each of them is an observation of a specified problem but all are coherent together. This approach needs a unique representation of the problem; all perspectives can be considered as views with several control features. By default, we propose five perspectives:

   (a) an eXtensible Markup Language (XML) perspective, where the XML files will be validated by the given HOπ Document Type Definition (DTD). The

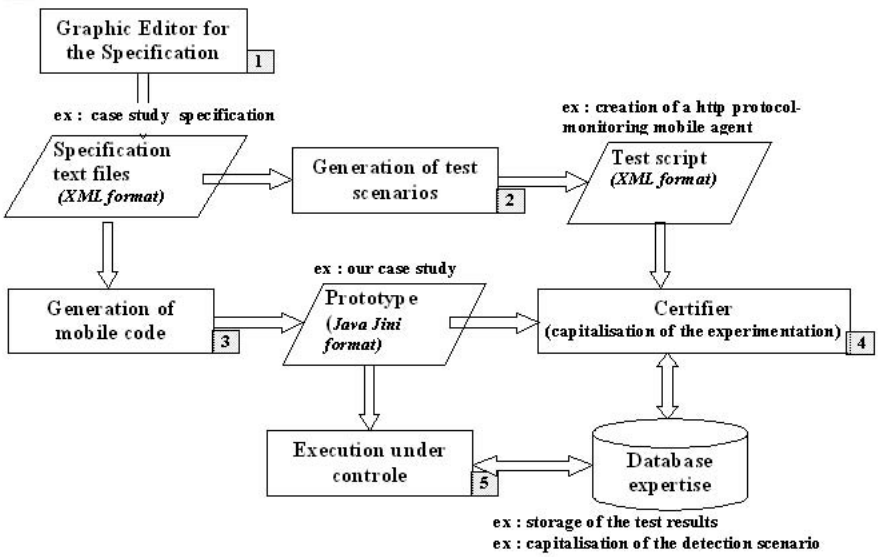


Figure 6.2: From Specification to Implementation

XML format is the intermediate format for all the tools which belong to the platform *HOPiTool*. It checks the input file and offers a myriad of functions (for instance, a fold editor). It also validates the syntax with the DTD of the π-Calculus language. This part of the tool corresponds to a pilot of a Java code generator from an XML input file which formally describes a π-Calculus language specification. It is used to create the charts. Any change of a chart has consequences on this intermediate representation. The structure of each perspective is based on the Model View Controller (MVC) design pattern.

(b) a π-calculus visualization respecting the style given by R. Milner. This perspective is the main one for the specifiers, or those who formally design protocols. The π-calculus syntax is preserved. This part is a pilot of a translation from this syntax into the XML syntax.

(c) an HTML visualization. This perspective stands for the documentation of the specified agent. It is used by people who learn or manage formal protocols. This perspective is only an observation and can be published on a web server for information. Moreover, HTML documents describing the documentation are generated by our tool, which takes into consideration the comments written within the specification.

(d) Because UML does not take into account mobile features, the UML perspective is complementary information which gives a solution to this descriptive

problem. We chose a way to describe mobility that combines collaboration diagrams with class diagrams. The collaboration diagram is the main support to show both the causality of the messages and the agents which exchange data. Class diagrams describe not only classes which correspond to the objects being exchanged or to the collaboration diagrams, but also classes which are entities of the sequence diagram. This perspective pilots the translation from UML into XML and is the basic one for a technical approach, but its descriptor is the most suitable for having an object-oriented prototype. Thus, it is the closest representation of a Java Jini source representation.

(e) The simulation perspective offers another data source. It provides observations of a particular execution of the Java Jini prototype. Also, it is the most technical perspective (the user has to provide the list of lookup services and the port numbers for the exchanger).

The context of a project involves the use of some techniques such as: the use of preferences, internationalization and also resource file management. Internationalization helps to maintain a single source code base for all language versions of our product. It also facilitates translations because all localizable resources are identified and isolated. These techniques are applied throughout the project. Furthermore, it concerns, not only the graphical interface, but also the simulation part: database access or log file access.
The major idea is to have a rigorous framework to control two important stages: the generation of test scenarios and the generation of code. As we can observe in Figure 6.2, the steps "Generation of test scenarios" and "Generation of mobile code" are independent, but both are based on the XML representation of the specification.

2. The generation of test scenarios
From XML files describing a π-calculus specification, which means a set of mobile agents communicating through the services they specify, the goal of this functionality is to generate test scenarios respecting the criteria given by the specifier. If we consider an agent specification as a set of automata, building a test generation corresponds to a tuple of paths, one path per automaton. The criteria are essential for reducing the number of tuples. When the set of criteria is not sufficient, it is possible to limit the length of an elementary path.
These criteria are meta rules useful for the generation of test sequences. The objective is to build tests which are used by the Certifier entity. For example, a criterion can express a property such that "navigates through all the agent declaration" to be sure that all declarations are taken into account for a test. Another

criterion can be to "navigate through the channel name, ch_1, ch_2" with a precise list of gates and, possibly, a number of exchanges for each of them. That property is essential to validate the higher-order communication, when an agent is exported to another one.

Each criteria corresponds to a set of Java assertions which are inserted into the generated code. Thus, the properties can be validated during simulations. Each simulation places its results (meaning the evaluation of the assertion) in specific log files (called by the name and the date of the simulation step). In other words, our first goal is to build structural "white boxes" and not functional tests, called, more commonly, "black boxes". Each output file describes a script of tests and the essential measurement points for the work executed by the Certifier.

3. The generation of the mobile code

   This functionality is initiated by the graphic interface and is used to obtain a code prototype from valid HOπ specifications. A set of source files from the XML files describing a π-calculus specification is generated in order to obtain a prototype. The aim is to gain the ability to do simulations and to validate the prototype through the tests we defined before. A specification consists of agents declarations. Each construction of the higher-order π-calculus language has a code generation pattern. For instance, the operator | for the parallel constructor involves the creation of Thread, one per operand. When this operator is evaluated, the threads are started. Following the same scheme, each communication is also an object. Because communication is synchronous in a π-calculus language, the generated code has to ensure this property. This is the reason why a communication is an object which locks the sender or the receiver until both are ready to exchange. In addition to using the synchronous method of Java, this pattern also used a serialization interface. This perspective is visualized through the source code view.

   When the data exchange does not concern a value, but an agent, the mechanism is not the same and we have used another pattern adopted to the migration of code. In that case, this pattern encapsulates Jini techniques (see chapter 5.3). This means that a sender has to define a lookup service and to publish its new service with it. The receiver uses a broadcast to find out the lookup service where the desired service is published.

   The mobile aspects take shape through the use of the technical framework Java Intelligent Network Interface (Jini) from Sun. The fundamental element is the utilization of Remote Method Invocation (RMI), which allows us to consider the network to be object-oriented. The services and clients in a Jini architecture can be more easily integrated within the object interface. Jini also possesses identifi-

cations and collection services, constitutes an important feature when considering the development of a mobile agent system. We also defined several subclasses of agent, because our framework can be used for different application domains, such as intrusion detection, data collection and log file collaboration. In the case of a security agent, the exchanges of data are encrypted and the author is checked via a signature for the reception of all data. Java toolkit contains a framework Java Cryptography Environment (JCE) from Sun, which handles the security questions.

4. The certifier
   This entity takes a Java prototype as input but also an abundance of assertions. Each assertion is evaluated over the prototype in order to determine if the prototype validates the criteria. Because a specific log file is generated for each simulation, the certifier module is called only after simulations are realized. Its work can be considered as a post-mortem analysis. The format of a log file is also based on an XML representation and contains information about the step when the assertions are evaluated (not only the result of this evaluation, but also the state of the current agent and its context). This perspective is an important metric for the specifier and some changes in the initial specification can be decided after this step. It is visualized through the simulation view.
   We can distinguish two cases: the prototype satisfies the test and this simulation will enrich the database or the prototype does not satisfy the test, an anomaly is lifted and a log file describes the context of the anomaly. The analysis is not included in this project.

5. Execution under control
   As it is essential to catch a lot of behavior, every simulation is saved through the use of watch points. This kind of observation is placed by the specifier or can be placed automatically at each call of agent. This source of data is also used by the certifier module. This data is saved in XML files and some metrics can be computed. This perspective is the closest to the simulation visualization.

The technical description of HOPiTool Version 1.0 is given in the appendix A. We would next like to show, on the basis of a case study, how our *HOPiTool* creates agents that are able to communicate.

6.4 Applying Simulation to our SLP Case Study

We now turn to the practical contribution in this chapter - the practice of the *HOPiTool*. *HOPiTool* is intended to provide the necessary facilities for agent implementation based on the formal agent model described previously. As shown above, the implementation platform provides the standard computer technologies, such as the Jini middleware and the Java Virtual Machine (JVM), for agent implementation.
Let us now consider the case study and its higher-order π-calculus specification described in section 4.2. In this specification, the mobility is described as the exchange of agents over several channels. XML is used in a first step in order to describe the whole system of our case study. The XML data uses a DTD in Source Code 21 (we show here only a portion of the DTD), which defines the HOπ syntax with all its elements. Each element has attributes that catch essential information used for the synthesis of code or a graph, such as the depth of a control structure. A parser will check the correctness of the XML file regarding the HOπ syntax with all its elements.

Source Code 21 — *DTD: The HOπ Syntax*

```
<?xml version="1.0" encoding="UTF-8"?>
<!ELEMENT SPECIFICATION (#PCDATA|AGENT)*>
<!ATTLIST SPECIFICATION
    name CDATA #REQUIRED
>
<!ELEMENT AGENT (#PCDATA|ARGUMENT|DEF)*>
<!ATTLIST AGENT
    name CDATA #REQUIRED
    argNumber CDATA #IMPLIED
>
<!ELEMENT DEF (#PCDATA|CALL|CHOICE|LABEL|PARALLEL|RECEIVE|SEQUENCE|SEND|ZERO)*>
<!ELEMENT LABEL EMPTY>
<!ATTLIST LABEL
    name CDATA #REQUIRED
    depth CDATA #REQUIRED
    param CDATA #IMPLIED
>
<!ELEMENT CHOICE (#PCDATA|LABEL|PARALLEL|SEQUENCE|RECEIVE|SEND|INTERNAL_ACTION|
    MATCH|ZERO)*>
<!ATTLIST CHOICE
    alternative CDATA #REQUIRED
```

```
>
<!ELEMENT PARALLEL (#PCDATA|CHOICE|SEQUENCE|LABEL|MATCH|INTERNAL_ACTION|ZERO|
    CALL|RECEIVE|SEND)*>
<!ATTLIST PARALLEL
    flow CDATA #REQUIRED
>
```

In a first step, using the defined DTD, we validate the case study through its appropriate XML file. The second step consists of a translation to π notation using the *HOPiTool*. The screenshot 6.3 below shows the graphical representation of the HOπ view. The

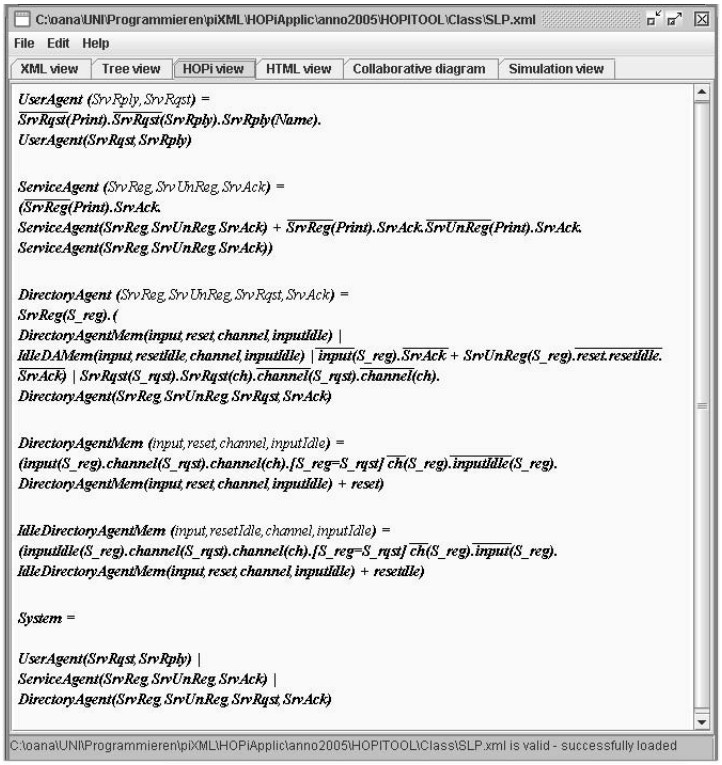


Figure 6.3: HOπ View Screenshot

prototype is also able to generate an HTML view (Figure 6.4) after the translation of the XML file, in order to deliver a documentation view. The last view shows a collaboration view of the same example. The third step is the code generation of the mobile agents (Figure 6.6). For each specified agent, a Jini/Java code will be generated automatically as the corresponding documentation file. As we mentioned before, the

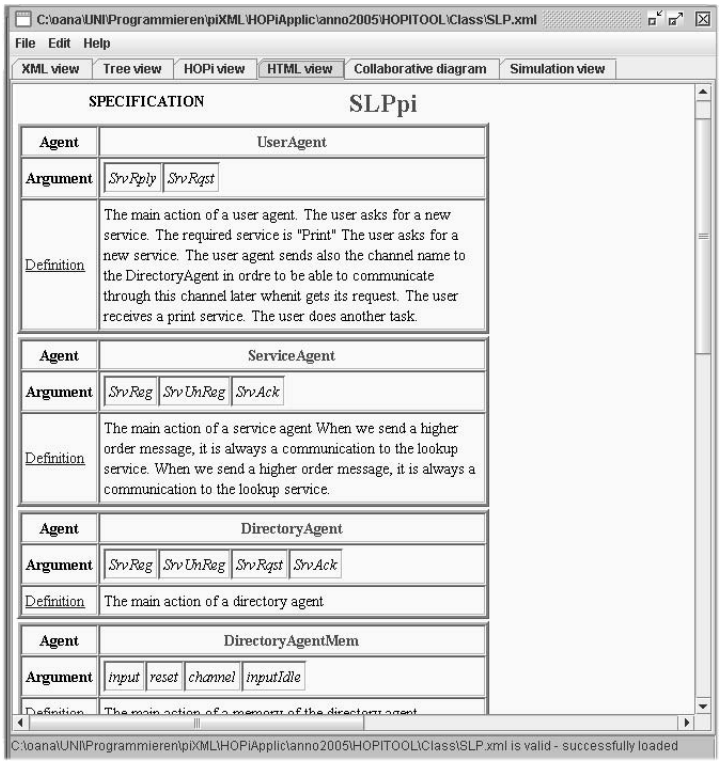


Figure 6.4: HOPiTool Screenshot

Jini middleware is used in order to resolve the problem of network administration by providing an interface where different components of the network can join or leave the network. Our tool has its own lookup service where user mobile services are published. Also, a simulation can be launched when the code generation is completed. The results are displayed in the simulation view. Because services can be bugged, the simulation view represents not only the results of mobile agents, but also a browser of all the services, their properties and their leases. In this case, a simulation is an evolution of the state of this simulation view. Since agents only interact with each other through asynchronous message passing, the service provided by an agent through Jini is designed as an interface to let other agents send asynchronous messages to that agent. The agent who sends out the messages then becomes the service consumer. The whole mechanism is explained in diagram 6.6. After Lookup initializes the first time, the *DirectoryAgent* and the *ServiceAgent* will publish their location in order to be found through unicast by other agents and clients. The *UserAgent* is the client that searches a service. For this purpose it will connect to the Lookup and get the available *DirectoryAgent*. Now it can begin its request within the *DirectoryAgent* for a service called "print". The

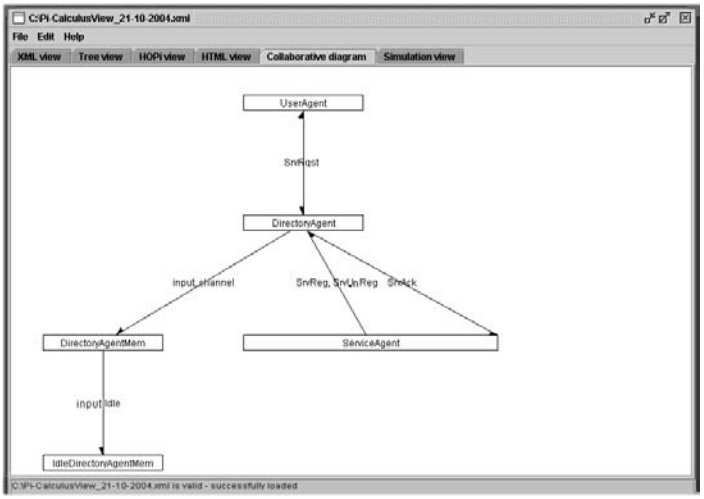


Figure 6.5: HOPiTool Screenshot

DirectoryAgent creates a new Lookup, where the *ServiceAgent* will register its service "print". After the *DirectoryAgent* gets the service, it will send it to the *UserAgent*, which, at the end, will be able to perform it. The Lookup and the new Lookup play the role of the DirectoryAgentMem and IdleDirectoryAgentMem.

6.5 Summary

Although a number of mobile-oriented systems have been built in the past few years, there is very little work on bridging the gap between theory, systems, and application. The purpose of this chapter is to use the mobile-oriented higher-order π-calculus model, which is a formal model, as a specification for the development of mobile agents. We thus bring formal methods directly into the design phase of the agent development. With the automatic generation of code for the *UserAgent*, *ServiceAgent* and the *DirectoryAgent* and the use of Jini, we are able to simulate mobile agents in a network to test and validate their behavior. The possibility of integrating an external model checker (for example the UPPAAL) into our system exists. With this model we will be able to both verify and validate mobile systems simultaneously, thereby employing the most efficient means in proving their correctness.

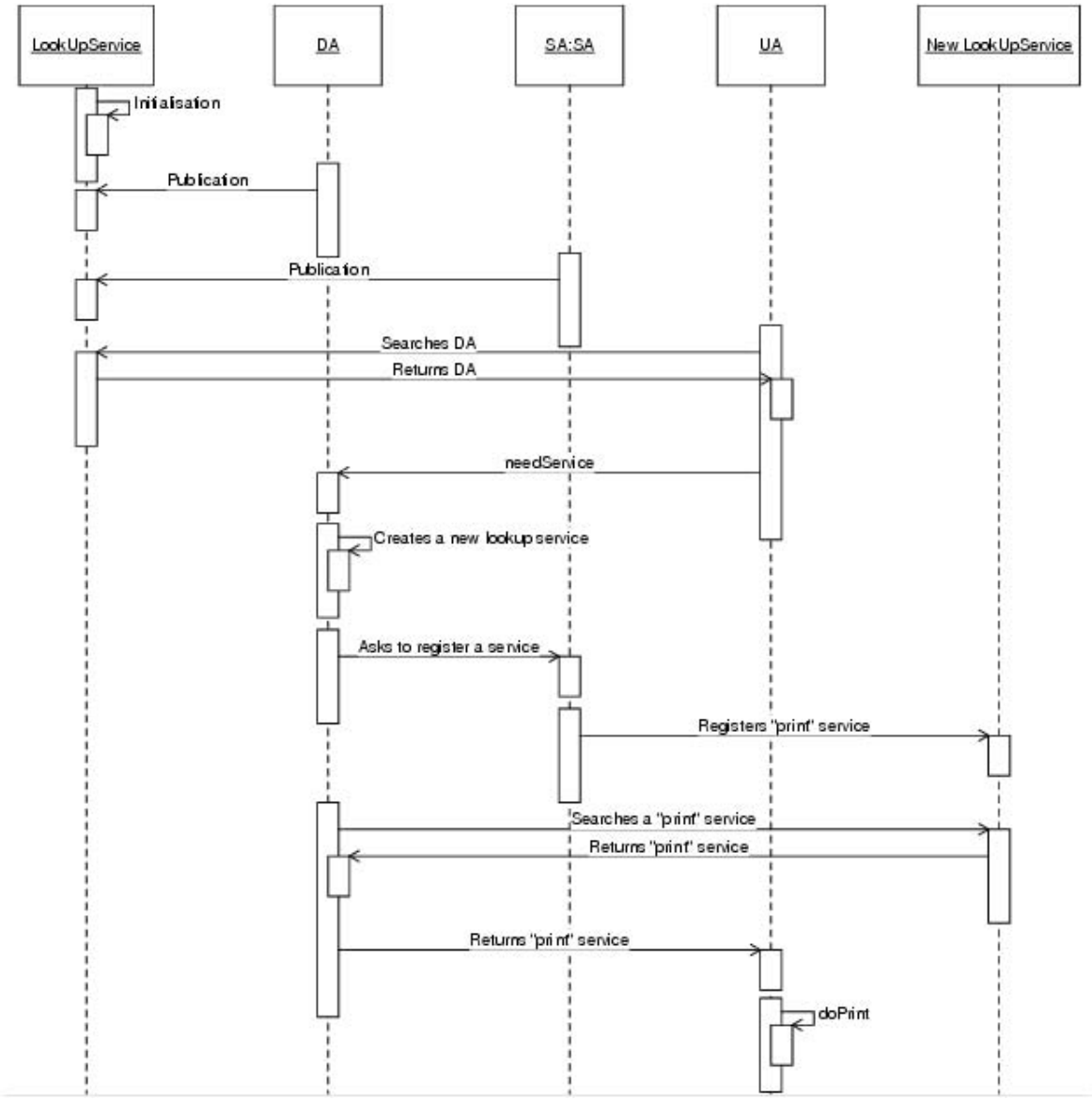


Figure 6.6: SLP Sequence Diagram

Chapter 7

Conclusion and Future Work

The addition of mobility could be considered a contribution to parallel systems in the same way that parallelism was to sequential programming. Initially, the introduction of mobility satisfied a need (for instance, a security need: the code migrates and not the data, or a reliability need: a client/server relation is replaced by a client/service relation, where the service comes to the client). Later, it was possible to introduce the notion of mobility by default within every problem and conceive it such that all data transfer would be replaced by an agent movement. This ensured independence: actually, this agent can transfer data very well (as it did before), but it can also transfer the processing of the requested data. We can notice that the introduction of mobility to these kinds of examples can bring more generality compared to the solution we have now. This new proposal will not make the use of older ones obsolete, but will bring new solutions. The difficulty of manipulating the notion of mobility, as well as in a specification case and in a coding case, shows the need of tools such as HOπTool. A further approach, concerning the use of programming patterns, such as IncaX [`http://www.incax.com`], can also be utilized, where it is possible to draw the mobility graph and to ask the tool to generate the Jini route in the same manner as for an automaton. This approach (GUI Builder) will surely be developed as soon as the designer can describe his route differently from an oriented graph, considering the movement cost as a property.

The work presented here deals with the modeling, validation and simulation of mobile agents. The contribution of this thesis was to provide a theoretical framework for the concept of mobile agents, and a prototype for their development and simulation allowing further the verification of properties by UPPAAL (which can be integrated within the prototype). After introducing the mobile agents with their mobility paradigms,

we described some selected successful national and international projects. Interesting discussion points were also given concerning the advantages and the disadvantages of mobile systems. Furthermore, the higher-order *pi*-calculus was introduced and its arity extended. We have shown that this language is consistent with the notion of mobility and is sufficient in order to model mobile agents. Finally, we have developed the prototype called HOPiTool based on this specification language.

Before starting the implementation of the prototype, we exposed three different methods for modeling a mobile agents system. The concluding discussion helped us in the selection of the most appropriate solution, which, in our case, was the use of Jini/Java. The architecture of this prototype has been designed in an effort to exploit the extensions and improvements. In addition, this prototype can be adapted to other programming environments which support mobile agents.

In subsequent future work, after improving the simulation view, security issues can also be taken into account. Another improvement is to add a higher-order unification mechanism to the HOPiTool. This is an important and difficult task due to the lack of research available on the implementation of higher-order algorithms. The linking of a model checker to our prototype constitutes another important and future development. The presented tool UPPAAL can, for example, be linked to the HOPiTool in order to verify properties of the designed mobile system. The verification together with the simulation, will improve the correctness of the developed system. The prototype also lends itself as an important aide for system designers, who wish to develop mobile agents. Through the combination of the specification and simulation views, this task is easier to accomplish and bugs can be detected more easily. The work we have proposed allows for the verification (together with the validation and the simulation) of the mobile systems designed using our HOPiTool. The results we obtain using this prototype depend on the reliability of the model-checker chosen. The improvement of the selected technologies such as Jini (current version 2.2) presents new ways of dealing with mobile systems. The absence of RMI within the latest version of Jini makes the new model protocol independent. Ultimately, the designer will have less and less manual work to do.

The originality of this research is based the fact that the modelling of a mobile agent system does not occur in any ad-hoc manner, but, instead, on corresponding theoretical investigations using a process algebra called π-calculus. There is currently no similar ongoing research that employs π-calculus in order to model mobile agent systems.

Appendix A

Technical Description of HOPiTool Version 1.0

The *HOPiTool* prototype implements five packages:

1. ***hopitool.gui*** package contains all the viewers which belong to the graphical part of the tools. The graphical user interface (called GUI) has several features:
   - each window created with *HOPiTool* has six tabs:
     - a tab for the XML view of the specification,
     - a tab for the Tree view, generated using the DOMViewer from the Batik-framework provided by Apache [`http://xml.apache.org/batik/`]. Batik is a Java technology based toolkit for applications or applets that use images in the Scalable Vector Graphics (SVG) format for various purposes, such as viewing, generation or manipulation.
     - a tab for the textual higher-order π-calculus specification,
     - a tab for the HTML view (or web documentation) of the specification,
     - a tab for the agent diagram (or UML collaboration diagram),
     - a tab for the simulation view described by the input specification
   - some dialog boxes are used for diagnostics such as:
     - a dialog for each previous tab,
     - a help manual for developers,
     - a start guide for the specifier.

This package is divided into several sub-packages which correspond to the views. Also, it is easier to add another one. In the current version, the GUI (see figure A.1) is used to display some information about the specification. In the current version only the XML view allows interactions from users. The modifications are propagated on to the other views. When the user clicks on a tab, the update of the corresponding view is computed. A given specification is observable only in a *HOPiTool* window. In the next version each view will support interactions from the user.

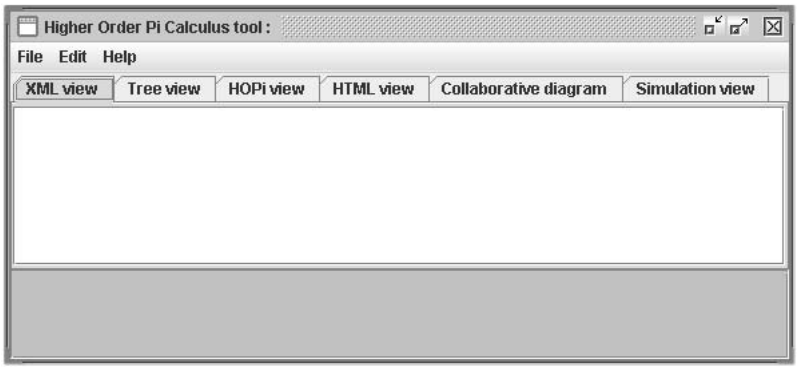


Figure A.1: HOPiTool Screenshot

2. ***hopitool.lang*** package contains contains the set of definitions for keywords used in the formal language higher-order π-calculus. It contains all patterns of code generation. This means that the Java/Jini semantic of higher-order π-calculus is defined within that package. Thus, it is easy to add some new features to that language such as: new operators (bounded iteration, oriented communication. etc). For each of the structures, there is a Java class. The design of these packages follows a classical approach. Also Factory patterns are used for the creation of the object, Template patterns are used for the implementation of code generation. Each class contains a method called *"generatedCode"* which describes the code generation step. The input specification file is an XML file and the default handler is defined in the file *PiCalculusHandler.java*.

3. ***hopitool.model*** represents the interface between View and Model. Since each HOπ specification is available as a XML file, we had to transform it into an internal java object in order to work with it. For this purpose we used the JAXB Framework. It generates automatically for each XML element java classes (for Example, all classes were automatically generated in hopitool.model.jaxb). Un-

fortunately, the possibilities to work with automatically generated java classes are limited, because some methods are missing. Concerning the use of JGraph, we needed information about the connections between objects, which are not delivered by the automatically generated methods (through JAXB). Because of that, we introduced additional proxy classes (in hopitool.model.proxy), that implement the missing methods. This method of implementation would help us to add new features to the simulation view in future work. The external interface (therefore the one used for the editor) is the class $ModelResource$ on the higher level.

4. ***hopitool.parsing*** package contains all the classes used for the analysis and the management of the symbol table. Some classes are used for a validation check.

5. ***hopitool.unification*** package contains all the classes used for the unification feature of the higher-order π-calculus language. It is used in several contexts:

   - during a communication between a send action and a receive action.
   - during the call of a process between a call action and a declaration of a process.

   This current unification is defined as a first-order mechanism. This means that agent definition or communication can have deeply structured parameters, yet the level of that structure is not bounded and the identifier of the non-atomic term is not free and closed within the sub-term. A declaration is realized:

   - with the use of a restriction operation which can be considered as a declaration statement,
   - with the use of parameters in the declaration of a process,
   - with the receive statement which involves the use of a variable for the input data.

   In a language like higher-order π-calculus language, a free variable is an identifier which supports a global renaming. It is the opposite of a bound variable which cannot support a local renaming. There is an important difference between a modifier (free or bound) and the value which is associated during the execution step. All variables have a values but the modifier (free or bound) is essential for the unification step. This operation appears during a communication and a call to an agent.
   A free variable is used to define the identifiers which occur in :

   - the parameters of an agent declaration,

- the right part of a reception statement,
- a restriction of the scope of identifiers.

One of the next objectives is to add a higher order unification mechanism under some restrictions [54], which will allow the computation of a unification of the name of a process.

6. ***hopitool.security*** package contains a specific approach of mobile agents and stresses the security features. All exchanges are encrypted. It is a specialization of hopitool.lang package.

7. ***hopitool.communication*** package describes some communication schemes. A lot of specifications use a communication graph, which is predefined or, more often, a scheme is well known in the domain of the specifier. For instance, a circular circuit or a multicast scheme are considered to be predefined in a framework. Some patterns are also used to allow a developer to build a new scheme such as a cube graph.

In summary, the important technical features are the Java 1.4.2 Swing Framework, the inputs and outputs are written in XML (with an external validation in the form of a DTD file), Java Intelligent Network Interface Jini 2.0., Remote Method Invocation (RMI) and Oracle 9i (used to manage the data). Other types of database can be used as Oracle features are not essential in our project. We can replace it by MySql or some other free database which completely respects the SQL language. We assume the use of a more powerful database for the more concrete studies, such as real protocols or when the specification describes more than one hundred agents. Our *HOPiTool* reads and generates Jini code from the XML file. The XML file is read and parsed using Java Architecture for XML Binding (JAXB) from Sun [`http://java.sun.com/xml/jaxb/`]. A DTD document describes how these XML files should be structured and states that an XML agent file contains a series of agents which play together in a mobile system. This DTD file comes from the definition of the higher-order π-calculus. Each agent has a name, a number of arguments and a definition of its form and its objective. We have defined each agent with the corresponding attributes and arguments. Based on this XML description of our mobile system, and with help of our *HOPiTool*, we are able to generate Jini code for a further simulation of communicating agents in a network.

MY PUBLICATIONS

(a) Andreea Barbu. *"Une Spécification d'ordre Supérieur π-Calculus d'un Agent Mobile"*, In Congrès Francophone MAJECSTIC'03, October, 2003.

(b) Andreea Barbu. *A tool for Supporting the Development of Correct Mobile Applications Based on Higher-Order π-Calculus*, In ESM'2004 European Simulation and Modelling Conference, UNESCO, Paris, France, October, 2004.

(c) Andreea Barbu and Fabrice Mourlin. *Mobile Properties and Temporal Logic*, In First Eurasian Conference on Advances in Information and Communication Technology, Shiraz, Iran, pages 1-6, Austrian Computer Society, October, 2002.

(d) Andreea Barbu and Fabrice Mourlin. *From Higher-Order π-Calculus Specification to RMI Implementation*, In International Conference on Computer Science, Software Engineering, Information Technology, e-Business and Applications, CSITeA'03, Rio de Janeiro, Brazil, June, 2003.

(e) Andreea Barbu and Fabrice Mourlin. *Higher-Order π-Calculus Specification for a Mobile Agent in JINI*, In 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03), pages 250-256, ACIS, 2003.

(f) Andreea Barbu and Fabrice Mourlin. *From π-Calculus Specification to a Simulation of a Mobile Agent Using Jini*, In ESM 2004, 18th European Simulation Multiconference, Magdeburg, Germany, June, 2004.

BIBLIOGRAPHY

[1] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM, 1997.

[2] M. Abdalla, W. Cirne, L. Franklin, and A. Tabbara. Security issues in agent based computing. In *15th Brazilian Symposium on Computer Networks, São Carlos/Brasilien*, May 1997.

[3] M. Alfalayleh and L. Brankovic. An overview of security issues and techniques in mobile agents. In *Conference on Communications and Multimedia Security*, September 2004.

[4] K. Arnold, A. Wollrath, B. O'Sullivan, R. Scheifler, and J. Waldo. *The Jini Specification*. AW, 1999.

[5] A. Asperti and N. Busi. Mpn. Technical Report UBLCS-96-10, University of Bologna, 1996.

[6] B. Bauer, D. Bonnefoy, F. Bergenti, and R. Evans. The lightweight extensible agent platform. In *Proceedings of the Autonomous Agent Conference*, February 2001.

[7] B. Bauer and R. Höllerer. *Übersetzung Objektorientierter Programmiersprachen*. SV, 1998.

[8] J. Baumann. Mobile agents: Control algorithms. In *Lecture Notes in Computer Science*, volume 1658. SV, 2000.

[9] J. Baumann, K. Rothermel, and M. Strasser. Mole - concepts of a mobile agent system. *World Wide Web Journal, Science Publishers, Holland*, 1, 3, Baltzer:123 – 137, 1998.

[10] C. Bäumer and T. Magedanz. Grasshopper - a mobile agent platform for active telecommunication. In *IATA*, pages 19 – 32, http://link.springer.de/link/service/series/0558/bibs/1699/16990019.htm, 1999.

[11] F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. *Jade: A White Paper*. http://exp.telecomitalialab.com, September 2003.

[12] J. Bengtsson, P. Christensen, P. Jensen, K.-G. Larsen, F. Larsson, and P. Pettersson. *UPPAAL: a Tool Suite for Validation and Verification of Real Time Systems - User Guide Version 2*. Uppsala University, http://www.docs.uu.se/rtmv/uppaal/uppaal-guide.ps.gz, 1996.

[13] E. Best, W. Fraczak, R. P. Hopkins, H. Klaudel, and E. Pelz. M-nets: An algebra of high-level petri nets, with an application to the semantics of concurrent programming languages. *Acta Informatica*, 35:813 – 857, 1998.

[14] A. Bieszczad and T. White. Mobile agents for network management. Technical Report 1, IEEE Communications Surveys, http://www.comsoc.org/pubs/surveys, Fourth Quarter 1998.

[15] W. Binder and V. Roth. Secure mobile agent systems using java: Where are we heading? In *17th ACM Symposium on Applied Computing, Special Track on Agents, Interactions, Mobility, and Systems (SAC/AIMS), Madrid, Spain*. ACM, March 2002.

[16] A. Blass, Y Gurevich, and J. V. den Bussche. Abstract state machines and computationally complete query languages. Technical report, Microsoft, http://research.microsoft.com, December 1999.

[17] J. K. Boggs. Ibm remote job entry facility: Generalize subsystem remote job entry facility. Technical Report 752, IBM Technical Disclosure Bulletin, August 1973.

[18] G. Boudol. The π-calculus in direct style. *Higher-Order and Symbolic Computation*, 11:177 – 208, 1998.

[19] D. E. Brake, G. Emami, and GLOBAL INFOTEK VIENNA VA. Control of agent based systems (coabs) grid. Technical Report A522524, Storming Media, http://www.stormingmedia.us/52/5225/A522524.html, June 2004.

[20] J. Byassee. Unleash mobile agents using jini. In *WWDC in Moscone Center, San Francisco, CA*, pages 23 – 27, June 2003.

[21] L. M. Camarihna-Matos and H. Afsarmanesh. telecare: Collaborative virtual elderly support communities. In *Proceedings of the 1st Workshop on Tele-Care and Collaborative Virtual Communities in Elderly Care, TELECARE*, ISBN: 972-8865-10-4. ICEIS 2004, 2004.

[22] L. Cardelli. Obliq: A language with distributed scope. Technical Report 122, Digital Equipment Corporation Systems Research Center, June 1994.

[23] L. Cardelli and A. D. Gordon. Anytime, anywhere: Modal logics for mobile ambients. In *POPL'00*, pages 365 – 377. ACM Press, 2000.

[24] Patrick Chan. *The Java(TM) Developers Almanac 1.4, Volume 1.* Sun, http://www.jcp.og, 2002.

[25] K. Chandy and J. Misra. Parallel program design. In *AW*, 1988.

[26] H. Christensen. Cognitive (vision) systems. In *ERCIM News, Special: Cognitive Systems*, 2003.

[27] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *LNCS*, volume 131, pages 52–71, 1981.

[28] General Magic Inc. GMD Focus Crystaliz, Inc. and IBM Coorp. Mobile agent facility specification. Technical report, OMG, 1997.

[29] M. Dam. Proof systems for the π-calculus logics. Technical report, Dept. of Teleinformatics, Royal Institute of Technology, Sweden, 1996.

[30] M. Dam. Digital design and life-cycle management for distributed information supply services in innovation exploitation and technology transfer. Technical Report IST-1999-10092, Hellenic TeleCommunications and Telematics Applications Company, 2001.

[31] C.Perkins E. Guttman. *Service Location Protocol (SLP), Version 2.* Sun Microsystems, http://www.ietf.org/rfc/rfc2608.txt.

[32] E. Emerson and J. Halpern. Sometimes and not never revisited: On branching versus linear time. In *Proc. 10th Annual ACM Symp. on Principles of Programming Languages, Austin*, pages 127–140, 1983.

[33] E. Emerson and J. Srinivasan. Branching time temporal logic. In *LNCS*, volume 354, pages 123–172, 1983.

[34] C. Fournet et al. A calculus of mobile agents. In *7th International Conference on Concurrency Theory (CONCUR'96)*, 1996.

[35] G.-C. Roman et al. Mobile unity: Reasoning and specification in mobile computing. Technical Report 6, ACM Transactions on Software Engineering and Methodology, 1997.

[36] S. Green et al. *Software Agents: A Review.* Department of Computer Science, Trinity College, Dublin, Ireland.

[37] C. Fencott. *Formal Methods for Concurrency.* TCP, 1996.

[38] G. Ferrari, S. Gnesi, U. Montanari, M. Pistore, and G. Ristori. Verifying mobile processes in the hal environment. In *Computer Aided Verification (CAV'98) LNCS*, volume 1427, http://fmt.isti.cnr.it:8080/hal/, 1998.

[39] FIPA. *Foundation for Intelligent Physical Agents.* FIPA, http://www.fipa.org/, 1999.

[40] S. Fischmeister, G. Vigna, and R. Kemmerer. Evaluating the security of three java-based mobile agent systems. In *5th International Conference on Mobile Agents (MA 01), Atlanta, GA*, December 2001.

[41] Foundation for Abstract Architecture Specification. Intelligent physical agents (fipa). Technical report, http://www.fipa.org, 2001.

[42] C. Fournet and G. Gonthier. The reflexive cham and the join-calculus. In *Conference Record of POPL'96, The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of programming Languages, ACM Press, New York*, pages 372–385, 1996.

[43] C. Fournet and G. Gonthier. The reflexive chemical abstract machine and the join-calculus. In *Proc. of POPL, ACM Press*, pages 372 – 385, 1996.

[44] A. Fuggette, G. P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342 – 361, May 1998.

[45] M. Fukuda. *MESSENGERS: A Distributed Computing System Based on Autonomous Objects.* PhD thesis, University of Washington, Bothel, 1997.

[46] S. Fünfrocken and F. Mattern. Mobile agents as an architectural concept for internet-based distributed applications. In *KiVS 99, Kommunikation in Verteilten Systemen, R. Steinmetz, Ed., Informatik Aktuell*, pages 32 – 43. SV, 1999.

[47] M. Grabner, F. Gruber, L. Klug, and W. Stockner. Agent technology: State of the art. Technical report, Software Competence Center Hagenberg, August 2000.

[48] B. Gray. Soldiers, agents and wireless networks: A report on the actcomm scenarios and testbed. In *Proceedings of the 2000 Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, 2000.

[49] R.S. Gray, D. Kotz ad G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile-agent system. *VIGNA*, pages 154 – 187, 1998.

[50] W. Grosso. *Java RMI*. O'Reilly, 2001.

[51] O. Gutknecht and J. Ferber. Madkit: a generic multi-agent platform. In *Agents*, pages 78 – 79, 2000.

[52] Silvia Hagen. *Guide to Service Location Protocol*. ISBN 1-893939-359. Podbooks Com. Llc, http://www.srvloc.org, 1999.

[53] F. Hohl. *Sicherheit in Mobile-Agenten-Systemen*. PhD thesis, Fakultät Informatik, Universität Stuttgart, http://elib.uni-stuttgart.de/opus/volltexte/2001/893, 2001.

[54] G. Huet. Higher-order unification 30 years later. In *15th International Conference TPHOL*, volume 2410, pages 3–12, September 2002.

[55] Adobe Systems Inc. *PostScript Language Reference Manual*, addison wesley edition, 1985.

[56] W. Jansen and T. Karygiannis. Mobile agent security. In *PNIST Special Publication National Institute of Standard and Technology*, volume 19, 2000.

[57] N. Karnik. *Security in Mobile Agent Systems*. PhD thesis, Department of Computer Science, University of Minnesota, 1998.

[58] J. Keogh. *J2ME - The Complete Reference*. ISBN 0072227109. Osborne/McGraw-Hill, 2003.

[59] S. I. Kumaran. *Jini Technology*. ISBN 013033859. Prentice Hall, 2002.

[60] D. Lange and M. Oshima. *Programming and Deploying Java Mobile*. Addison-Wesley Longman, 1998.

[61] D. B. Lange and M. Ishima. *Programming and Deploying Java Mobile Agents with Aglets*. AW, 1998.

[62] D. B. Lange and M. Oshima. Seven good reasons for mobile agents. *ACM*, 42(3):88 – 89, 1999.

[63] L.Cardelli and A. D. Gordon. Mobile ambients. In *Foundations of Software Science and Computation Structures, Lisbon*, 1998.

[64] R. Lea, C. Jacquemont, and E. Pillevesse. Cool: System support for distributed object oriented programming. *ACM*, 36(9):37 – 46, November 1993.

[65] G. Di Marzo, M. Muhugusa, and C. F. Tschudin. A survey of theories for mobile. In *World Wide Web Journal, Special Issue on Distributed World Wide Web Processing: Applications and Techniques of Web Agents*, pages 139–153, http://gdimarzo.home.cern.ch/gdimarzo/webjournal.ps, 1998. Baltzer Science Publishers.

[66] P. J. McCann and G.-C. Roman. Mobile unity: A language and logic for concurrent mobile systems. Technical report, WUCS-97-01, Department of Computer Science, Washington University in St. Louis, December 1996.

[67] R. Milner. The polyadic π-calculus: a tutorial. ECS-LFCS-89-85 91–180, University of Edinburgh, 1991.

[68] R. Milner. *Communication and Concurrency*. PHL, 1998.

[69] R. Milner. *Communicating and Mobile Systems: The pi-Calculus*. Cambridge University Press, 1999.

[70] R. Milner, J.Parrow, and D. Walker. A calculus of mobile processes i and ii. *Information and Computation*, 100(1):1–77, 1992.

[71] Dejan Milojicic. Mobile agent applications. *IEEE Concurrency*, 7(3):89 – 90, July 1999.

[72] Y. Minsky, R. Renesse, F. B. Schneider, and S.D. Stoller. Cryptographic support for fault-tolerant distributed computing. In *Distributed Computing , Proceedings of the Seventh ACM SIGOPS European Workshop*, pages 109 – 114, 1996.

[73] U. Montanari and M. Pistore. History-dependent automata. Technical report, Technical Report Dipartimento di Informatica, Universita di Pisa, Italy, 1997.

[74] T. Mota, S. Gouveris, G. Pavlou, A. Michalas, and J. Psoroulas. Quality of service management in ip networks using mobile agent technology. In *Proceedings of the IEEE/ACM International Workshop on Mobile Agents for Telecommunication Applications (MATA'2002)*, October 2002.

[75] R. De Nicola, G. Ferrari, and R.Pugliese. Klaim: A kernel language for agents interactions and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[76] Sun Org. *Enterprise JavaBean Technologies*. Inc., California, USA, http://java.sun.com/products/ejb.

[77] Sun Org. *Surrogate project*. The Jini Community, http://wwws.sun.com/ software/jini/, 2000.

[78] Sun Org. *Jini Network Technology*. Inc., California, USA, 2001.

[79] Sun Org. *Jini Network Technology: Datasheet*. Inc., California, USA, 2001.

[80] J. Parrow. *Handbook of Process Algebra*. Elsevier, ISBN: 0-444-82830-3, 2001.

[81] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. In *Proc. of LICS, IEEE Computer Society Press*, pages 176 – 185, 1998.

[82] J. Parrow and B. Victor. The fusion calculus: Expressiveness and symmetry in mobile processes. *LICS'98, Computer Society Press*, pages 176–185, 1998.

[83] R. Pascotto. *AMASE: A Complete Agent Platform for the Wireless Mobile Communication Environment*. T-Nova Deutsche Telekom Innovationsgesellschaft mbH Berkom, http://www.cordis.lu/infowin/acts/analysys/products/thematic/agents/ch3/amase.htm, 1998.

[84] H. Peine and T. Stolpmann. The architecture of the ara platform for mobile agents. In *First International Workshop on Mobile Agents MA'97*, volume 1219, pages 50–61, Berlin, Germany, april 1997. SV.

[85] L. M. Peña. *Techniques for the Development of Fault-Tolerant Distributed Application on Corba and Java-RMI Architectures*. PhD thesis, Computer Science School. Universidad Complutense de Madrid, http://grasia.fdi.ucm.es/sensei/docs/sensei.pdf, May 2002.

[86] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical report, CSCI 476, Computer Science Department, Indiana University, 1997.

[87] V. Roth. Obstacles to the adoption of mobile agents. In *5th IEEE International Conference on Mobile Data Management (MDM 2004), 19-22 January 2004, Berkeley, CA, USA*, pages 296–297, 2004.

[88] T. Sander and C. Tschudin. Towards mobile crypthography. In *IEEE Symphosium on Security and Privacy, Oakland, CA*, 1998.

[89] D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, University of Edinburgh, 1992.

[90] D. Sangiorgi and D. Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.

[91] P. Sewell. Applied pi - a brief tutorial. Technical Report 498, University of Cambridge, 2000.

[92] A. Shafi, U. Farooq, S. L. Kiani, M. Riaz, A. Shehzad, A. Ali, I. Legrand, and H. B. Newman. Diamonds - distributed agents for mobile & dynamic services. *CoRR*, cs.DC/0305062, 2003.

[93] Roger Spottiswoode. *Tomorrow Never Dies: James Bond 007*, 1997.

[94] Markus Strasser. *Fehlertoleranz Mobiler Agenten*. PhD thesis, Universitaet Stuttgart Department of Computer Science, 2002.

[95] SysteMATech. http://www.systematech.org/index.php?Project.

[96] W. Theilmann. *Themenspezifische Informationssuche im Internet mit Hilfe mobiler Programme*. PhD thesis, Fakultät Informatik, Universität Stuttgart, http://elib.uni-stuttgart.de/opus/volltexte/2000/703, 2000.

[97] W. Thielmann and K. Rothermel. Hawk: Harvesting the widely distributed knowledge.

[98] V. Vasudevan and S. Landis. Malleable services. In *Proceedings of the 34th Hawaii International Conference on System Sciences*, October 2001.

[99] B. Victor and F. Moeller. The mobility workbench - a tool for the π- calculus. In *Proc of CAV*, volume 818, pages 428 – 440. Springer, 1994.

[100] G. Vigna. Mobile agents: Ten reasons for failure. In *Proceedings of MDM Berkeley, CA*, pages 298–299, January 2004.

[101] J. Waldo. The end of protocols. Technical report, Sun Org., http://java.sun.com/developer/technicalArticles/jini.

[102] M. Weiss, C. Busch, and W. Schrter. *Multimedia Arbeitsplatz der Zukunft - Assistenz und Delegation mit mobilen Softwareagenten*. Talheimer Verlag, 2003.

[103] T. White and B. Pagurek. Towards multi-swarm problem solving in networks. In *Proc. of the 3rd Intl Conf. on Multi-Agent Systems (ICMAS 98)*, July 1998.

[104] P. Wojciechowski and P. Sewell. Nomadic pict: Language and infrastructure design for mobile agents. *IEEE Concurrency*, 8(2), 2000.

INDEX

Glossary

Bluetooth Wireless personal area network (PAN) standard that enables data connections between electronic devices such as desktop computers, wireless phones, electronic organizers and printers in the 2.4 GHz range at 720kbps within a 30-foot range. Bluetooth depends on mobile devices equipped with a chip for sending and receiving information. *Page 21.*

EJB-container A container that implements the EJB component contract of the J2EE architecture. This contract specifies a runtime environment for enterprise beans that includes security, concurrency, life-cycle management, transactions, deployment, naming, and other services. An EJB container is provided by an EJB or J2EE server. *Page 136.*

GSM Global System for Mobile communications: Digital cellular radio technology. Operates in the 900 MHz waveband. *Page 21.*

HTTP Hypertext Transfer Protocol: protocol used to browse World Wide Web sites on the Internet (Computers). *Page 127.*

IDL Interface Definition Language that facilitiates interfacing between servers and IDL compliant client computers. For example, a Java IDL enables Java to communicate with non-Java objects on networks. *Page 113.*

IIOP Internet Inter-ORB Protocol: A protocol used for communication between CORBA object request brokers. *Page 127.*

IrDA Infrared Data Association: A membership organization founded in 1993 and dedicated to developing standards for wireless, infrared transmission systems between computers. With IrDA ports, a laptop or PDA can exchange data with a desktop

computer or use a printer without a cable connection. Like a TV remote control, IrDA requires line-of-sight transmission. IrDA products began to appear in 1995. *Page 21.*

J2EE Java 2 Platform, Enterprise Edition: An environment for developing and deploying enterprise applications. The J2EE platform consists of a set of services, application programming interfaces (APIs), and protocols that provide the functionality for developing multi-tiered, Web-based applications. *Page 146.*

JAAS Java Authentication and Authorization Service: (1) In J2EE technology, a standard API for performing security-based operations. JAAS implements a Java version of the standard Pluggable Authentication Module (PAM) framework. (2) A package through which services can authenticate and authorize users while enabling the applications to remain independent from underlying technologies. *Page 136.*

JavaBean A specification developed by Sun Microsystems that defines how Java objects interact. An object that conforms to this specification is called a JavaBean.

JBoss The JBoss/Server is the leading Open Source, standards-compliant, J2EE based application server implemented in pure Java. *Page 137.*

Jini Jini is defined as an infrastructure and programming model which allow devices to connect with each other to create an instant.

JLS Jini lookup service, the central component of Jini's runtime infrastructure, offers Jini clients a flexible and powerful way to find Jini services. It enables service providers to advertise their services and enables clients to locate and enlist the help of those services. *Page 126.*

JNDI Java Naming and Directory Interface: An extension to the Java platform that provides a standard interface for heterogeneous naming and directory services. *Page 137.*

JNI Java Native Interface: A programming interface that allows Java code to interoperate with functions that are written in other programming languages.

JTA Java Transaction API: An API that allows applications and J2EE servers to access transactions. *Page 136.*

JTS Java Transaction Service. Specifies the implementation of a transaction manager which supports JTA and implements the Java mapping of the OMG Object Transaction Service (OTS) 1.1 specification at the level below the API. *Page 136.*

JVM Java Virtual Machine. (Computers) software that serves as interpreter between Java bytecode and a specific operating system (allows Java applications to run on any platform without changing the code). *Page 33.*

PDA Personal Digital Assistant: A small, handheld wireless device capable of storing and transmitting pages, data messages, voice calls, faxes and e-mails. A typical PDA can function as a cellular phone, fax device, Web browser and personal organizer. Unlike portable computers, most PDAs began as pen-based devices that used a stylus rather than a keyboard for input. Many PDAs subsequently incorporate handwriting recognition features. Some PDAs can also react to voice input by using voice recognition technologies. Most PDAs are available in either a stylus or keyboard version. PDAs are also called palmtops, hand-help computers, Personal Information Managers (PIMs), and pocket computers. *Page 21.*

PostScript A page description language (PDL) developed by Adobe Systems. Widely supported by both hardware and software vendors, it represents the current standard in the market. John Warnock and Chuck Geschke of Adobe both worked for Xerox at the Palo Alto Research Center where PDLs were invented and set up their company to commercially exploit the concepts they had helped develop. *Page 49.*

reggie is the Lookup Service implemented by Jini. *Page 131.*

RPC Remote Procedure Call: An easy and popular paradigm for implementing the client-server model of distributed computing. In general, a request is sent to a remote system to execute a designated procedure, using arguments supplied, and the result returned to the caller. There are many variations and subtleties in various implementations, resulting in a variety of different (incompatible) RPC protocols. *Page 113.*

rsh Remote Shell. *Page 49.*

software Object modelling language specifically designed for the creation of **micromodels**. *Page 32.*

SSL An SSL digital certificate is an electronic file that uniquely identifies individuals and servers. Digital certificates allow the client (Web browser) to authenticate the server prior to establishing an SSL session. Typically, digital certificates are signed by an independent and trusted third party to ensure their validity. *Page 61.*

UDP User Datagram Protocol: protocol with no connection required between sender and receiver that allows sending of data packets on the Internet (though unreliable because it cannot ensure the packets will arrive undamaged or in the correct order). *Page 127.*

UMTS Universal Mobile Telecommunications System: UMTS is the European standard of the IMT2000 family of third generation mobile cellular standards (3G). In addition to the telephony service, UMTS will in particular allow the provision of multimedia services (data, images, sound) at data transfer rates of 144 kbps (vehicles), 384 kbps (pedestrians) and 2 Mbps (buildings). *Page 21.*

UNIX A computer operating system, originally developed at ATT Bell Laboratories, that is compatible with a wide range of computer systems. Ultrix, Solaris, AIX, HP/UX, BSD, Linux, and SystemV are among its numerous descendants. *Page 49.*

WAP The Wireless Application Protocol (WAP) is an open, globally recognized, protocol specification that empowers mobile services subscribers to use wireless devices to easily access and interact with information and services similar or identical to those available on the 'Web' or 'Net'. Overcoming the constrains of (relatively) slow and intermittent nature of wireless links for mobile communications, together with the limited screen size and computing power of mobile devices, is the central goal of WAP technology. *Page 21.*

WML Wireless Mark-up Language, or WML, is similar to HTML in many ways and serves a similar purpose. However, WML has been specially designed and optimized to take into account the constraints of mobile communications and mobile subscriber equipment such as cell phones and PDAs. *Page 21.*

WWW World Wide Web or W3 or The Web: a distributed HyperText-based information system conceived at CERN to provide its user community an easy way to access global information. *Page 22.*

X.509 A widely used standard for defining digital certificates. X.509 is actually an ITU Recommendation, which means that it has not yet been officially defined or approved for standardized usage. As a result, companies have implemented the standard in different ways. For example, both Netscape and Microsoft use X.509 certificates to implement SSL in their Web servers and browsers. But an X.509 Certificate generated by Netscape may not be readable by Microsoft products, and vice versa. *Page 61.*

XML An acronym for eXtensible Markup Language: XML is a flexible way to create common information formats and share both the format and the data on the World Wide Web, intranets, and elsewhere. For example, computer makers might agree on a standard or common way to describe the information about a computer product (processor speed, memory size, and so forth) and then describe the product information format with XML. Such a standard way of describing data would enable a user to send an intelligent agent (a program) to each computer maker's Web site, gather data, and then make a valid comparison. XML can be used by any individual or group of individuals or companies that wants to share information in a consistent way. Currently a formal recommendation from the World Wide Web Consortium (W3C). XML is similar to the language of today's Web pages, HTML. Both XML and HTML contain markup symbols to describe the contents of a page or file. HTML, however, describes the content of a Web page (mainly text and graphic images) only in terms of how it is to be displayed and interacted with. *Page 112.*

VITA

Name:	Andreea Barbu
Born:	September 16th, 1974
Nationality:	German
Education:	
2000 - 2005	PhD Thesis at the department of computer science of ParisXII University, France and Carl von Ossietzky, University of Oldenburg, Germany.
1994 - 2000	Student of Computer Science with applications to Psychology at the University of Oldenburg, Germany. Final examination: Diploma in Computer Science.
1994	High school Diploma (Abitur) at the Alvin Lonke Gymnasium, Bremen, Germany
Working Experience:	
2000 - 2005	Scientific assistant at the department of Computer Science at the University ParisXII Creteil, France.
1998 - 2000	Assistant as an undergraduate at the department of Computer Science at the Carl von Ossietzky University in Oldenburg
Scholarships:	
2000 - 2003	Supported by the ministry of research in France.