Developing Component-Based Applications with a Data-Centric Approach and within a Service-Oriented P2P Architecture: Specification, Analysis and Middleware

Ayoub Ait Lahcen

UNIVERSITÉ DE NICE-SOPHIA ANTIPOLIS
École Doctorale
Sciences et Technologies de
l'Information et de la Communication

UNIVERSITÉ MOHAMMED V AGDAL-RABAT
Centre d'Études Doctorales
Sciences et Technologies de Rabat

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice-Sophia Antipolis
et de l'Université Mohammed V Agdal
Spécialité : INFORMATIQUE

Présentée et soutenue par

Ayoub AIT LAHCEN

Developing Component-Based Applications with a Data-Centric Approach and within a Service-Oriented P2P Architecture: Specification, Analysis and Middleware

(Développement d'Applications à Base de Composants avec une Approche Centrée sur les Données et dans une Architecture Orientée Service et Pair-à-Pair : Spécification, Analyse et Intergiciel)

soutenue le 15 décembre 2012

Jury

Président :	Driss ABOUTAJDINE	PES à l'Université Mohammed V Agdal-Rabat
Rapporteurs :	Gilles ROUSSEL	Prof. à l'Université Paris-Est Marne-la-Vallée
	Mahmoud NASSAR	PH à l'ENSIAS
Examinateurs :	Mireille BLAY-FORNARINO	Prof. à l'Université de Nice-Sophia Antipolis
	Jacques PASQUIER	Prof. à l'Université de Fribourg
Co-encadrant :	Salma MOULINE	PH à l'Université Mohammed V Agdal-Rabat
Co-directeur :	Didier PARIGOT	Chargé de Recherche, INRIA Sophia Antipolis

To my Parents

To my Sister

To my Brother

Acknowledgements

This doctoral thesis has been prepared in joint guardianship with Mohammed V Agdal University (at LRIT laboratory) and Nice Sophia Antipolis University (at INRIA Sophia Antipolis).

My first thank goes to my thesis advisors Prof. Driss ABOUTAJDINE (LRIT laboratory) and Dr. Didier PARIGOT (INRIA Sophia Antipolis). Without the various help they provided me, the achievement of this dissertation would have never been possible. I thank them for their support, patience, and useful advices. I'm deeply grateful to Prof. Driss ABOUTAJDINE for encouraging me during the final stages of my Master's project to think about doing a PhD. His enthusiasm to propel scientific research in Morocco is something that I admire and hope to replicate throughout my career. I would also like to thank him for serving as my thesis committee chair. Likewise, I owe a great debt to Dr. Didier PARIGOT, he provided me with all that a PhD candidate could ever need. I greatly appreciate his willingness to guide me in improving this work, especially through the countless stimulating discussions we had together. His professionalism and friendship will always be appreciated.

I would like to thank Prof. Salma MOULINE (LRIT laboratory), my co-advisor, for giving me useful suggestions and comments for the improvements of this work. I would also like to highlight that she gave me great confidence by choosing me to teach, since my first PhD year, Master's courses in Component-Based Software Development.

I wish to express my thanks and gratitude to Prof. Gilles ROUSSEL (Paris-Est University), Prof. Mahmoud NASSAR (ENSIAS), Prof. Mireille BLAY-FORNARINO (Nice Sophia Antipolis University) and Prof. Jacques PASQUIER (Fribourg University) for accepting to be members of my thesis committee. Their valuable feedback and inspiring comments helped me to improve this dissertation in several ways.

Special thanks go to Dr. Pascal DEGENNE (CIRAD), Dr. Danny LO SEEN (CIRAD), Dr. Remi FORAX (Gaspard Monge Institut) and Dr. Olivier CURE (Gaspard Monge Institut). I have collaborated with them on the STAMP project (a French research project aiming at developing a new modelling language for describing environmental landscapes and their dynamics). It has been a great pleasure to work with them and I always have a feeling that what I learnt

and took away from STAMP project is much more than what I gave through my contribution. The work related to this project is reported in Chapter 8.

To so many people in INRIA Sophia Antipolis who directly or indirectly helped me and made it a great experience, I'm deeply grateful. Special thanks to my team Zenith and particularly to its head Dr. Patrick VALDURIEZ. To a pleasant group of colleagues with whom I shared lively lunch discussions about numerous subjects, thank you for creating a cheerful atmosphere and an endless succession of bursts of laughter: Alexandre CARABIAS, Anca BELME, Hubert ALCIN, Dr. Alain DERVIEUX, Dr. Valérie PASCUAL and Dr. Laurent HASCOET.

I thank all PhD students and staffs in LRIT laboratory, it was nice to be among them during the months I stayed each year in Morocco. I'm pleased to have Brahim AKBIL as a colleague and as a dear friend. My deepest gratitude and thanks to him for all the support he made available throughout my doctoral studies. He is always helpful and enjoyable. Thank you Brahim for this invaluable friendship. A particular thank goes to my Master's classmates whom are now PhD students in LRIT laboratory: Abdelkaher, Ahmed, Laila, Said. Thank you for all the great moments we shared together. I wish you all the best.

I would like now to acknowledge and thank those who have provided me with their unlimited support, encouragement, understanding and patience. They have been always there for me and have helped me in every possible way. My immense gratitude to my parents, my brother Soufiane, my sister Dina and her husband Nour Eddin, and of course, to their little boy Yassir who has been a source of joy and great relaxation that made me forget the stresses of work. A big thank to all of you for your never-ending love.

Résumé

Le développement d'applications avec une architecture Pair-à-Pair (P2P) est devenu de plus en plus important en ingénierie du logiciel. Aujourd'hui, un grand nombre d'organisations de tailles et secteurs différents compte d'une manière croissante sur la collaboration entre multiples acteurs (individus, groupes, communautés, etc.) pour accomplir des tâches essentielles. Ces applications P2P ont généralement un comportement récursif que plusieurs approches de modélisation ne peuvent pas décrire et analyser (ex. les approches basées sur les automates à états finis). Un autre challenge qui concerne le développement d'applications P2P est le couplage fort entre la spécification d'une part, et les technologies et protocoles sous-jacents d'autre part. Cela force les développeurs à faire des efforts considérables pour trouver puis comprendre des informations sur les détails de ces couches basses du P2P. De plus, ce couplage fort oblige les applications à s'exécuter dans des environnements figés. Par conséquent, choisir par exemple un autre protocole pour répondre à un nouveau besoin à l'exécution devient une tache très difficile. Outre ces points, les applications P2P sont souvent spécifiées avec une faible capacité à déléguer des traitements entre les pairs, et se focalisent surtout sur le partage et le stockage de données. Ainsi, elles ne profitent pas pleinement de la puissance de calcul et de traitement offerte par le réseau P2P sous-jacent.

Dans cette thèse, nous présentons une approche qui combine les principes du développement orienté composants et services avec des techniques issues des Grammaires Attribuées et d'analyses de flot de données (techniques utilisées surtout dans la construction de compilateurs) afin de faciliter la spécification, l'analyse et le déploiement d'applications dans des architectures P2P. Cette approche incorpore : i) Un langage formel nommé DDF (de l'anglais Data-Dependency Formalism) pour spécifier les applications et construire leurs graphes de dépendances de données. Un graphe de dépendances de données est nommé DDG (de l'anglais Data-Dependency Graph) et est défini pour être une représentation abstraite de l'application spécifiée. ii) Une méthode d'analyse qui utilise le graphe de dépendances de données pour inférer et calculer diverses propriétés, y compris certaines propriétés que les model-checkers ne peuvent pas calculer si le système présente un comportement récursif. iii) Un intergiciel nommé SON (de l'anglais Shared data Overlay Network) afin de développer et d'exécuter des applica-

tions dans une architecture P2P sans faire face à la complexité des couches sous-jacentes. Cela grâce essentiellement au couplage faible (par une approche orientée services) et à la fonctionnalité de génération de code automatique.

Mots-clés : Spécification Formelle, Analyse Formelle, Dépendances de Données, Développement de Logiciels à Base de Composants (CBSD), Architecture Orientée Services (SOA), Pair-à-Pair (P2P).

Abstract

Developing Peer-to-Peer (P2P) applications became increasingly important in software development. Nowadays, a large number of organizations from many different sectors and sizes depend more and more on collaboration between actors (individuals, groups, communities, etc.) to perform their tasks. These P2P applications usually have a recursive behavior that many modeling approaches cannot describe and analyze (e.g., finite-state approaches). Another challenging issue in P2P application development is the tight coupling between application specification and the underlying P2P technologies and protocols. This forces software developers to make tedious efforts in finding and understanding detailed knowledge about P2P low level concerns. Moreover, this tight coupling constraints applications to run in a changeless runtime environment. Consequently, choosing (for example) another protocol at runtime to meet a new requirement becomes very difficult. Besides these previous issues, P2P applications are usually specified with a weak ability to delegate computing activities between peers, and especially focus on data sharing and storage. Thus, it is not able to take full advantages of the computing power of the underlying P2P network.

In this thesis, we present an approach that combines component- and service-oriented development with well-understood methods and techniques from the field of Attribute Grammars and Data-Flow Analysis (commonly used in compiler construction) in order to offer greater ease in the specification, analysis and deployment of applications in P2P architecture. This approach embodies: i) A formal language called DDF (Data-Dependency Formalism) to specify applications and construct their Data-Dependency Graphs (DDGs). A DDG has been defined to be an abstract representation of applications. ii) An analysis method that uses DDG to infer and compute various properties, including some properties that model checkers cannot compute if the system presents a recursive behavior. iii) A component-based service middleware called SON (Shared-data Overlay Network) to develop and execute applications within a P2P architecture without the stress of dealing with P2P low level complexity. Thanks to SON's automatic code generation.

Keywords: Formal Specification, Formal Analysis, Data-Dependency, Component-Based Software Development (CBSD), Service-Oriented Architecture (SOA), Peer-to-Peer (P2P).

Associated Publications

This thesis is partially based on the following peer-reviewed publications:

International journals

- Ayoub Ait Lahcen, Didier Parigot, Salma Mouline. *"A Data-Centric Formalism with Associated Service-Based Component Peer-to-Peer Infrastructure"*. Information and Software Technology. Under review (modifications have been requested).

- Pascal Degenne, Danny Lo Seen, Didier Parigot, Remi Forax, Annelise Tran, Ayoub Ait Lahcen, Olivier Curé and Robert Jeansoulin. *"Design of a Domain Specific Language for modelling processes in landscapes"*. Ecological Modelling. Volume 220 (24), pages 3527-3535, December 2009.

International conferences

- Ayoub Ait Lahcen, Didier Parigot. *"A Lightweight Middleware for developing P2P Applications with Component and Service-Based Principles"*. The 15th IEEE International Computational Science and Engineering. December 2012, Paphos, Cyprus.

- Ayoub Ait Lahcen, Didier Parigot, Salma Mouline. *"Defining and Analyzing P2P Applications with a Data-Dependency Formalism"*. The 13th International Conference on Parallel and Distributed Computing, Applications and Technologies. December 2012, Beijing, China.

- Ayoub Ait Lahcen, Didier Parigot, Salma Mouline. *"Toward Data-Centric View on Service-Based Component Systems: Formalism, Analysis and Execution"*. Work in Progress Session of the 20th EUROMICRO International Conference on Parallel, Distributed and Network-based Processing. February 2012, Munich, Germany.

- Pascal Degenne, Ayoub Ait Lahcen, Olivier Curé, Remi Forax, Didier Parigot, Danny Lo Seen. *"Modelling with behavioural graphs. Do you speak Ocelet?"*. International Congress on Environmental Modelling and Software. July 2010, Ottawa, Ontario, Canada.

- Olivier Curé, Rémi Forax, Pascal Degenne, Danny Lo Seen, Didier Parigot, Ayoub Ait Lahcen. *"Ocelet: An Ontology-based Domain Specific Language to Model Complex Domains"*. The First International Conference on Models and Ontology-based Design of Protocols, Architectures and Services. June 2010, Athens, Greece. (Best paper award)

- Christophe Proisy, Elodie Blanchard, Ayoub Ait Lahcen, Pascal Degenne, Danny Lo Seen, *"Toward the simulation of the Amazon-influenced mangrove-fringed coasts dynamics using Ocelet"*. International Conference on Integrative Landscape Modelling. February 2010, Montpellier, France.

- Ayoub Ait Lahcen, Pascal Degenne, Danny Lo Seen, Didier Parigot, *"Developing a service-oriented component framework for a landscape modeling language"*. The 13th International Conference on Software Engineering and Applications. November 2009, Cambridge, massachusetts, USA.

Contents

List of Figures

List of Tables

Part I:

Opening

Chapter 1

Introduction – in French

Sommaire

1.1 Vue d'ensemble

Le développement d'applications avec une architecture Pair-à-Pair (P2P) est devenu de plus en plus important en ingénierie du logiciel. Aujourd'hui, un grand nombre d'organisations de tailles et secteurs différents compte d'une manière croissante sur la collaboration entre multiples acteurs (individus, groupes, communautés, etc.) pour accomplir des tâches essentielles. Une architecture P2P est un concept où chaque entité agit à la fois comme serveur et client

dans un réseau P2P [Schollmeier, 2001]. Cela est complètement différent des architectures Client/Serveur où une entité peut agir uniquement en tant que serveur ou client, sans être capable de jouer les deux fonctions en même temps. Ainsi, dans une architecture P2P, les rôles des différentes entités sont approximativement égaux et chaque entité fournit des services aux autres en tant que pair.

Dans les systèmes logiciels, en particulier ceux qui sont déployés sur des architectures P2P, les données échangées sont nécessaires pour accomplir des tâches de traitement et acheminer des interactions entre les différentes entités du système. Néanmoins, la conception de systèmes logiciels se focalise généralement sur l'ordonnancement des activités de traitement et néglige le flot de données. Une approche centrée sur les données fournit une méthode différente de voir et de concevoir des applications logicielles. Elle permet de s'intéresser de plus près au flot et à la transformation de données tout le long du cycle de vie de l'application.

Dans ce contexte, nous avons défini un graphe de dépendances de données nommé DDG (de l'anglais Data-Dependency Graph). Ce graphe a été choisi pour former une représentation abstraite de l'application, et ce, pour les raisons suivantes. Premièrement, elle ne représente qu'un modèle de flux de données (imposé par la dépendance entre les données). Deuxièmement, DDG expose un niveau de détail suffisant pour effectuer des analyses de flot de données.

Dans cette thèse, nous présentons une approche qui combine les principes du développement orienté composants et services [Szyperski, 1998] [Huhns and Singh, 2005] avec des techniques issues des Grammaires Attribuées (AGs) [Paakki, 1995] et d'analyses de flot de données [Aho et al., 2006] (techniques utilisées surtout dans la construction de compilateurs) afin de faciliter la spécification, l'analyse et le déploiement d'applications dans des architectures P2P. Cette approche incorpore : i) Un langage formel nommé DDF (de l'anglais Data-Dependency Formalism) pour spécifier les applications et construire leurs graphes de dépendances de données. ii) Une méthode d'analyse qui utilise le graphe de dépendances de données pour inférer et calculer diverses propriétés, y compris certaines propriétés que les model-checkers ne peuvent pas calculer si le système présente un comportement récursif. iii) Un middleware nommé SON (de l'anglais Shared data Overlay Network) afin de développer et d'exécuter des applications dans une architecture P2P sans faire face à la complexité des couches sous-jacentes du P2P.

Le middleware SON est utilisé comme un environnement d'exécution qui gère les besoins liés au P2P (comme la gestion de mécanismes de communications, de files d'attente ou de diffusions de messages). La gestion de ces aspects est facilitée grâce essentiellement au couplage faible (par une approche orientée services) et à la fonctionnalité de génération de code automatique. Cette génération de code réduit et simplifie les tâches de développeurs d'applications et leur permet de se concentrer davantage sur la logique métier.

Le formalisme DDF fournit un ensemble d'opérations nécessaires pour spécifier et analyser des applications P2P. DDF peut être considéré comme un formalisme minimal et léger pour les raisons suivantes. D'une part, l'objectif de DDF est de construire formellement un graphe de dépendance qui expose le bon niveau d'abstraction pour effectuer des analyses de flots de données. D'autre part, DDF n'est pas destiné à écrire le détail du code métier ou à être un langage de programmation généraliste. Il a été plutôt pensé suivant les principes des langages dédiés (DSL – de l'anglais Domain-Specific Language) [Mernik et al., 2005]. DDF est fortement inspiré des caractéristiques des grammaires attribuées, notamment parce que ces dernières sont capables non seulement de construire un graphe de dépendance similaire, mais aussi de capturer naturellement un comportement récursif complexe (ce qui est très fréquent dans le cas d'applications P2P – voir Section 2.2.1) que de nombreuses autres approches ne peuvent pas décrire ou analyser.

L'environnement d'exécution du middleware SON peut être vu comme un ensemble de composants en interaction. Ces interactions sont dues à des envois et des réceptions de services. Lorsqu'un service est reçu ou envoyé, des données peuvent être échangées (par exemple, les paramètres de service, le résultat du service ou des données propres aux composants). La propagation d'appels de services entre composants peut dépendre de données transportées par un certain service appelé antérieurement. Par conséquent, notre approche a pour vocation d'étendre la spécification de services (qui définit souvent seulement les entrées et les sorties de composants) par la notion de dépendance. Cette notion capture non seulement les dépendances entre les services, mais aussi les dépendances entre les données échangées (requises et fournies). En définissant cette notion, il nous sera possible d'inférer le flot de données d'une composition/un assemblage de composants et de construire un graphe de dépendance de données de l'ensemble.

Une fois que le graphe de dépendance de données est construit à partir de la spécification DDF, nous pouvons vérifier ou inférer plusieurs propriétés en analysant le flot de données. Dans le Chapitre 6, cela est illustré à travers deux exemples. Le premier exemple montre comment résoudre le problème de détection d'interblocage (deadlock en anglais) par une recherche de circularité dans le graphe. Le deuxième exemple montre comment calculer la relation de dominance (entre données), en cherchant l'ensemble des dominateurs de chaque nœud du graphe. D'autres analyses (inspirées de travaux faits autour des GAs, comme ceux de [Parigot et al., 1996] et [Jourdan and Parigot, 1990]) peuvent être effectuées. Par exemple, en analysant l'ordre d'évaluation de données, il sera possible de déterminer formellement quels services dans un système peuvent être exécutés d'une manière parallèle ou incrémentale.

En plus de la construction du graphe de dépendance d'un système, notre formalisme DDF est capable de capturer naturellement un comportement récursif grâce à une spécification orientée règles. Il est bien connu dans la théorie des langages formels que de tel comportement ne peut pas être capturé par des automates à états finis (FSA, de l'anglais Finite-State Automata) [Aho et al., 2006]. Cela implique que les approches basées sur des FSA ne peuvent pas être utilisées pour décrire ou analyser des applications logicielles qui intègrent de tels comportements, en particulier, dans le contexte du développement à base de composants, où un grand nombre d'approches de modélisation sont basées sur des FSA. Deux des approches à composants les plus connues, SOFA [Bures et al., 2008] et Fractal, [Bulej et al., 2008] soulèvent clairement ce problème. Par exemple, dans [Bures et al., 2008], les auteurs précisent : *"our approach cannot treat behavior that cannot be modeled by a regular language (e.g. recursion)"*. Ainsi, de telles approches à composants ne sont pas adéquates dans le contexte de construction d'applications P2P où le comportement récursif est très fréquent, comme expliqué dans la section suivante.

1.2 Motivations et problématiques

1.2.1 Spécificité des applications P2P

L'évolutivité et l'auto-organisation comptent parmi les propriétés importantes des applications P2P. Cela en raison du très grand nombre d'utilisateurs et de la spécificité de connexions entre

les différents nœuds (connexions bas débit, haut débit, non stable, etc.) [Ripeanu et al., 2002]. Pour soutenir l'évolutivité et l'auto-organisation dans ce type de réseau, un grand nombre d'algorithmes et de protocoles P2P ont été développés. Ceux-ci sont souvent exécutés d'une manière récursive. Considérons, par exemple, le calcul de la réputation[1] qui est un problème d'une grande importance dans les environnements P2P [Aberer and Despotovic, 2001] (un exemple simple qui justifie cette importance est le cas où, durant le téléchargement de fichiers avec un logiciel de partage de fichiers en P2P, nous espérons que seulement des pairs fiables soient choisis). Le calcul de la réputation repose sur une séquence de requêtes envoyées pour obtenir des informations concernant la fiabilité d'un pair A et les réponses correspondantes à ces requêtes. Ce calcul doit être effectué d'une manière récursive, car une réponse reçue d'un autre pair B résulte d'une requête concernant la fiabilité de B. En plus, pour que le calcul soit correct, toutes les réponses doivent être reçues dans le bon ordre, puisque la condition d'arrêt peut dépendre de cet ordre-là. Ces envois récursifs de requêtes/réponses peuvent être vus comme une suite de parenthèses bien formées si chaque requête est remplacée par une parenthèse ouvrante et si chaque réponse correspondante est remplacée par une parenthèse fermante. Par conséquent, l'ensemble de ces séquences bien parenthésés est langage de Dyck[2]. Par exemple, la séquence "(())()" est bien parenthésée, et est un mot de Dyck, alors que la séquence "())(" ne l'est pas. Dans la théorie des langages formels, Il est bien connu qu'un langage de Dyck n'est pas un langage régulier [Stanley, 2001]. Dès lors, il n'existe aucun automate à états finis qui reconnait un langage de Dyck.

Ce type d'envois récursifs de requêtes/réponses, présenté ci-dessus, peut être bien spécifié en termes de langages non contextuels ou d'automates à piles (discutés plus tard dans la Section 4.1). Cependant, il est très fréquent que les protocoles P2P présentent un comportement récursif plus complexe qui donne lieu à des structures contextuelles – des structures d'interactions qui ajustent leurs comportements en fonction du changement de contexte. Pour illustrer cela, considérons le cas où quatre pairs voisins échangent des informations selon une interaction correspondant à deux envois récursifs de requêtes/réponses entrelacés. Ce type d'interaction ($a^n b^m c^n d^m$) a une structure contextuelle et, par conséquent, il ne peut pas être spécifié par un

[1] Nous notons ici que le calcul de la réputation présente un cas particulier de la diffusion d'informations et qu'il peut être effectué en utilisant le protocole Gossip présenté plus tard dans la Section 2.2.4.

[2] Un langage de Dyck D est un sous-ensemble de $\{x, y\}^*$ tel que si x est remplacé par une parenthèse ouvrante et y par une parenthèse fermante, nous obtenons une séquence de parenthèses bien formées [Stanley, 2001].

langage non contextuel [Aho et al., 1986].

En se référant aux travaux de recherche sur les Grammaires Attribuées [Parigot et al., 1996], qui sont des langages sensibles au contexte, le comportement récursif des applications P2P peut être capturé en décrivant à la fois le flot de contrôle et le flot de données de chaque interaction. En plus, une fois spécifié, ce comportement peut être analysé en utilisant des algorithmes d'analyse de flot de données. En outre, la hiérarchie de Chomsky [Chomsky, 1956], qui est une classification des langages formels, assure les inclusions suivantes :

$$\textit{type-3} \subsetneq \textit{type-2} \subsetneq \textit{type-1} \subsetneq \textit{type-0}$$

avec :

type-3 : Langages réguliers (reconnus par des automates à états finis).

type-2 : Langages non contextuels (reconnus par des automates à pile).

type-1 : Langages contextuels (reconnus par des machines de Turing non-déterministes).

type-0 : Langages récursivement énumérables (reconnus par des machines de Turing).

1.2.2 Vers des analyses de flot de données pour les applications P2P

1.2.2.1 Les model-checkers et la spécificité des applications P2P

Le model-checking est une technique automatisée qui, étant donné un modèle à états finis d'un système et une propriété formelle, vérifie systématiquement si cette propriété est satisfaite pour (un état donné dans) ce modèle [Baier and Katoen, 2008]. Il explore tous les états possibles du système d'une manière exhaustive. Le model-checking a été utilisé avec succès dans des domaines différents tels que les systèmes embarqués, la conception de matériels informatiques et le génie logiciel. Malheureusement, tous les systèmes ne peuvent pas tirer profit de sa puissance. Une des raisons est que certains systèmes ne peuvent être spécifiés par un modèle à états finis, en particulier, dans le contexte d'applications P2P (comme c'est expliqué ci-dessus). Une autre raison est que le model-checking n'est pas adapté pour des applications qui manipulent intensivement des données (et qui sont souvent développées en utilisant le para-

digme P2P [Lee et al., 2007] [Ranganathan et al., 2002]). Le récent livre sur le model-checking [Baier and Katoen, 2008] justifie clairement pourquoi la vérification d'applications manipulant intensivement des données est extrêmement difficile. En fait, même s'il n'y a qu'un petit nombre de données, l'espace d'état à analyser peut être très grand. Les auteurs du livre considèrent même que ces deux raisons sont parmi les premières limitations du model-checking :

"The weaknesses of model checking :

- *It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.*
- *Its applicability is subject to decidability issues ; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.*
- *... "*

1.2.2.2 Vérification par analyse de flot de données

L'analyse de flot de données réfère à un ensemble de techniques qui infèrent des informations sur le flot de données le long des chemins d'exécution d'un système logiciel [Aho et al., 2006]. L'exécution d'un système logiciel peut être vue comme une série de transformations de l'état du système (constitué à partir de l'ensemble des valeurs de toutes les variables du système). Chaque exécution d'une instruction intermédiaire transforme un état d'entrée à un nouvel état de sortie. On dénote les valeurs de flot de données, respectivement, avant et après une instruction s par $INPUTS[s]$ et $OUTPUTS[s]$.

Pour analyser le comportement d'un système, il faut prendre en compte tous les chemins d'exécution possibles dans le graphe de flot de données. Ainsi, résoudre un problème d'analyse de flot de données revient à trouver une solution à un ensemble de contraintes (appelées équations de flot de données) sur les $INPUTS[s]$ et $OUTPUTS[s]$, pour toutes les instructions s. Il existe deux types de contraintes :

- Contraintes sémantiques : elles définissent la relation entre les $INPUTS[s]$ et $OUTPUTS[s]$, pour chaque instruction s. Cette relation est généralement présentée comme une fonction

de transfert f, qui prend la valeur de flot de données $INPUTS[s]$ avant l'instruction et produit une nouvelle valeur de flot de données $OUTPUTS[s]$ après l'instruction. Autrement dit, $OUTPUTS[s] = f_s(INPUTS[s])$.

- Contraintes du flot de contrôle : si un système comprend les instructions $s_1, s_2, ..., s_n$ dans cet ordre, alors, la valeur sortant de s_i est la même que celle entrant dans s_{i+1}. Autrement dit, $INPUTS[s_{i+1}] = OUTPUTS[s_i]$, pour tout $i = 1, 2, ..., n - 1$.

Par exemple, pour vérifier une propriété telle que la vivacité des variables, qui détermine si une variable est susceptible d'être utilisée dans un chemin du graphe de flot de données, nous définissons les contraintes de vivacité des variables (c.-à-d., définir les équations de flot de données spécifiant qu'une variable d est active (vive) en un point p si un chemin commençant en p contient une utilisation de d). Ces équations peuvent être résolues en utilisant un algorithme itératif. La convergence de cet algorithme est assurée par le théorème du point fixe [Kam and Ullman, 1976] qui garantit qu'une solution unique de type point fixe existe pour ces équations. Une fois calculée, la vivacité d'une variable est une information très utiles. Par exemple, après l'utilisation de la valeur d'une variable à un point donné de l'exécution, il n'est pas nécessaire de garder cette valeur en mémoire si elle n'est pas utilisée ultérieurement, le long d'un chemin d'exécution. Dans la Section 6.3.2, nous présentons un autre exemple (détection de dominance) qui illustre plus en détail les principes de l'analyse de flot de données.

Plusieurs d'autres propriétés peuvent être calculées à ce niveau d'abstraction (c.-à-d., le graphe de flot de données), y compris certaines propriétés que les model-checkers ne peuvent pas calculer si le système a un espace d'états infini (voir par exemple [Govindarajan et al., 1992]). En outre, un grand nombre d'algorithmes a été proposé dans la littérature pour calculer ces propriétés. Malheureusement, à ce jour, l'utilisation principale de ces algorithmes (et en général, l'analyse de flot de données) reste dans le contexte de la construction de compilateur, en particulier, pour les algorithmes des grammaires d'attribuées, qui sont utilisés pour des analyses sémantiques dans la plupart des compilateurs.

Notre motivation dans ce contexte est de tirer avantage de ces algorithmes et techniques qui ont déjà prouvés leurs efficacités pour faciliter la spécification et l'analyse d'applications dans des environnements P2P.

1.2.3 Une approche centrée sur les données pour systèmes à composants

Le développement à base de composants [Szyperski, 1998] est devenu de plus en plus important en génie logiciel. Cela est dû essentiellement au besoin d'utiliser les concepts de cette approche pour implémenter des services et augmenter le niveau d'abstraction en facilitant la réutilisation, l'extension, la personnalisation et la composition de services [Yang and Papazoglou, 2004]. Ainsi, les services sont encapsulés dans des composants avec des interfaces bien définies pour être réutilisés dans plusieurs nouvelles applications. Cependant, le flot de données qui permet aux services d'accomplir des activités de traitements et qui guide les interactions entre composants, n'est souvent pas pris en compte, voir même totalement négligé, alors que dans plusieurs domaines de recherche tels que le Grid Computing, l'Informatique Décisionnelle et le P2P, les données sont incorporées comme une part importante du développement de systèmes. Récemment, dans le domaine émergent du Cloud Computing, où tout est service, la gestion de données a fait l'objet d'une attention remarquable et d'un grand intérêt [Abadi, 2009], et cela ne peut que croître.

Notre motivation dans ce contexte du développement à base de composants est de permettre aux données et de leurs flots d'être facilement spécifiés, vus et analysés, en particulier dans des environnements P2P. Alors que la plupart des approches à composants actuelles se focalisent sur les aspects structurels et fonctionnels de la composition de composants, nous insistons sur le fait que la modélisation du flot et la dépendance entre données a le même degré d'importance. Essentiellement, parce que les interactions entre composants sont guidées et acheminées par les données échangées.

1.2.4 Exemple illustratif : la spécification du protocole Gossip

Afin de motiver et illustrer l'intérêt de notre approche, en particulier dans le contexte des applications P2P, nous expliquons notre formalisme à travers la spécification du protocole Gossip [Voulgaris et al., 2005] [Jelasity et al., 2007]. Le protocole Gossip, appelé également protocole épidémique, est un protocole bien connu dans la communauté du P2P. Il est utilisé principalement pour assurer une diffusion/dissémination fiable de l'information dans un système distribué, d'une manière très similaire à la propagation des épidémies dans des communautés bio-

logiques. Ce type de dissémination est un comportement commun dans diverses applications P2P, et selon [Jelasity, 2011], un grand nombre de protocoles distribués peuvent être réduits au protocole Gossip. Il existe différentes variantes du protocole Gossip. Toutefois, un template qui couvre un nombre considérable de ces variantes a été présenté par Jelasity dans [Jelasity, 2011]. Dans notre exemple, nous nous basons sur cette template présentée ci-dessous :

Algorithm 1 Squelette de l'algorithme Gossip *(d'après [Jelasity, 2011])*

```
loop
   timeout(T)
   node ← selectNode()
   send gossip(state) to node
end
procedure onPushAnswer(msg)
   send answer(state) to msg.sender
   state ← update(state, msg.state)
end
procedure onPullAnswer(msg)
   state ← update(state, msg.state)
end
```

Pour modéliser ce protocole Gossip, nous considérons un ensemble de nœuds qui s'activent périodiquement à chaque pas de temps T et disséminent ensuite des données dans le réseau en échangeant des messages. En fait, quand un nœud reçoit des données, il répond à l'expéditeur, puis propage à son tour les données dans le réseau (en pratique, les données sont envoyées à un sous-ensemble de nœuds sélectionnés selon un algorithme spécifique). En terme de services, un nœud est un composant qui a deux activités : *servir* et *consommer* des données. Il existe deux services d'entrée (de l'anglais input services) pour l'activité *servir* et de deux services de sortie (de l'anglais output services) pour l'activité *consommer*. Ces services sont décrits dans l'interface du nœud comme suit :

$$(\{answer(resp : String), gossip(info : String)\}_{in},$$
$$\{gossip(info : String), answer(resp : String)\}_{out})$$

Le service *gossip* est utilisé pour à la dissémination de données, alors que le service *answer* est utilisé pour l'envoi de la réponse à l'expéditeur. Le comportement des services d'entrée (l'activité *servir*) inverse tout simplement les mêmes étapes des services de sortie (l'activité

consommer). A partir de cette description de services, nous pouvons construire intuitivement un graphe de dépendance simple entre les services. En fait, les services de sortie d'un nœud $node_x$ sont connectés aux services d'entrée d'un autre nœud $node_y$, et ainsi de suite. Ce graphe représente une partie du flot de contrôle, mais il n'offre pas une information très explicite sur le flot de données. Nous ne pouvons pas savoir quelles sont les dépendances entre les services et entre les données dans un nœud.

Pour compléter l'interface d'un nœud x avec une description à la fois du flot de contrôle et celui de données, notre formalisme DDF spécifie le comportement sous la forme de règles :

$$
\begin{array}{lll}
r_1: & timeout(T) & \rightarrow (gossip(state_x), node_y) \\
r_2: & (gossip(state_y), node_y), [onPush] & \rightarrow (answer(state_x), node_y) \\
r_3: & (gossip(state_y), node_y), [onPull] & \rightarrow \\
r_4: & (answer(state_y), node_y) & \rightarrow
\end{array}
$$

r_1 indique que le service interne *timeout* active $node_x$ à chaque pas de temps T, puis envoie la donnée $state_x$ au $node_y$ par l'intermédiaire du service *gossip*. r_2 indique que $node_x$ reçoit la données $state_y$ du $node_y$, puis répond $node_y$ en lui envoyant la donnée $state_x$ par l'intermédiaire du service *answer* si la condition *onPush* est satisfaite. *onPush* est une pré-condition (pour simplifier les choses, nous ignorons dans cet exemple ces pré-conditions). r_3 indique que $node_x$ reçoit la donnée $state_y$ du $node_y$ par l'intermédiaire du service *gossip*. r_4 indique que $node_x$ reçoit la donnée $state_y$ du $node_y$ par l'intermédiaire du service *answer*.

En introduisant ces règles, le système peut être vu comme un ensemble de composants où chacun de ces derniers a des entrées (partie gauche des règles) et des sorties (partie droite des règles). Les entrées reçoivent des données portées par des services, et après un traitement, les données résultantes peuvent être envoyées à travers les sorties. Ainsi, nous pouvons extraire un graphe de dépendances entre données de l'ensemble du système, en connectant les graphes partiels de dépendance entre données de chaque composant utilisé dans ce système.

1.2.5 Le besoin d'un runtime orienté composants, services et P2P

Des technologies très performantes ont été développées dans le contexte du P2P. Cependant, la plupart de ces technologies ne sont pas exploitées dans le processus de développement d'applications en raison des limitations des approches de spécification utilisées. Une de ces limitations

est le couplage fort entre la logique métier et les protocoles sous-jacents du P2P. Cela force les développeurs à faire des efforts considérables pour trouver puis comprendre les détails de ces protocoles. De plus, ce couplage fort contraint les applications à s'exécuter dans des environnements figés. Ainsi, choisir (par exemple) un autre protocole pour répondre à un nouveau besoin d'exécution devient très difficile. Une autre limitation est que les applications P2P sont généralement spécifiées avec une faible habilité à déléguer des activités de calcul/traitement à d'autres pairs, et se focalisent en particulier sur le partage et le stockage de données. Par conséquent, elles ne profitent pas pleinement de la puissance de calcul qu'offre le réseau P2P sous-jacent.

Une Architecture Orientée Services (SOA) est une forme d'architecture pour concevoir et développer des applications avec un couplage faible. Son but principal est non seulement de fournir un modèle d'intégration flexible (en réduisant les dépendances), mais aussi un haut niveau d'abstraction (en encapsulant les détails). Dès lors, la capacité d'applications à évoluer et à s'adapter aux nouveaux besoins augmente. Dans la littérature, SOA est souvent couplée avec les principes du développement à base de composants pour proposer des intergiciels. Cependant, la plupart de ces derniers ne sont pas adaptés aux applications P2P. Une des raisons est que ces intergiciels reposent sur des registres de services centralisés. Cet élément central dans une architecture SOA peut provoquer un goulot d'étranglement et causer le blocage de tout le système en cas de sa défaillance. Cela présente des risques de fiabilité et limite l'évolutivité d'applications. Une deuxième raison est que ces intergiciels utilisent des protocoles de communication qui ne sont pas adaptés aux environnements P2P. Par exemple, une application P2P n'est pas obligée de fonctionner en utilisant un Système de Noms de Domaine (DNS, de l'anglais Domain Name System) parce que les pairs n'ont pas toujours une adresse IP permanente.

Dans cette thèse, nous présentons le middleware SON. SON étend les principes de SOA, ainsi que ceux du développement à base de composants pour développer et déployer des applications dans une architecture P2P d'une manière facile et efficace.

SON assiste les développeurs d'applications en leur fournissant un mécanisme de génération automatique de code. Ce code généré s'occupe de plusieurs aspects liés au P2P comme la gestion de la communication, les files d'attente ou la diffusions de messages. En fait, l'utilisateur de SON implémente seulement le code métier correspondant aux services déclarés.

Ensuite, l'outil de génération de code génère les composants correspondants ainsi que leurs conteneurs associés. Le conteneur du composant incorpore toutes les ressources nécessaires pour adapter le code implémenté à l'environnement d'exécution P2P.

SON peut être considéré comme un middleware générique et léger (avec l'ensemble des opérations nécessaires qui doivent être présentes pour développer des applications P2P à base de composants et services), et ce, pour la raison suivante. Dans la plupart des cas, les challenges auxquels les systèmes P2P font face peuvent se réduire à un seul problème : *"How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy ?"* [Balakrishnan et al., 2003], SON a unifié la notion de *publish/subscribe* : il utilise une table de hachage distribuée (DHT, de l'anglais Distributed Hash Table) [Rhea et al., 2004] non seulement pour publier et consommer des données, mais aussi pour permettre de publier, découvrir et déployer dynamiquement des services.

1.3 Contributions

Le travail présenté dans cette thèse comporte quatre contributions majeures. Avant de donner un résumé de chacune de ces quatre contributions, nous présentons d'abord dans la sous-section suivante des idées clés sous-jacentes à ces dernières.

1.3.1 Des idées clés dans nos contributions

Faire attention aux dépendances : Toute spécification informatique est exprimée dans un langage contenant des dépendances entre les données et entre les différentes étapes/pas de la spécification. Les développeurs d'applications accordent généralement peu d'attention à ces dépendances, en particulier, aux dépendances "non-directes" qui peuvent être très difficiles à localiser sans une analyse automatisée. Dans de nombreux cas, une bonne gestion de ces dépendances améliore considérablement le fonctionnement d'applications. Par exemple, en réduisant le nombre de dépendances, plusieurs optimisations peuvent être réalisées (comme réduire le temps d'exécution ou l'espace mémoire utilisé).

Séparer ce qui est calculé du comment il est calculé : Grosso modo, les dépendances peuvent

être détectées et ajustées à trois niveaux : au moment de la spécification, au moment de la compilation et à celui de l'exécution. Dans notre cas, nous nous intéressons aux dépendances au moment de la spécification et avec l'idée de séparer, autant que possible, ce qui est calculé du comment il est calculé. L'avantage de cela vient du fait que non pas une seule mais plusieurs implémentations peuvent être synthétisées à partir de la spécification, et ce, grâce à l'analyse de dépendances entre données (par exemple, analyser comment évaluer les données d'une manière incrémentale, partielle ou parallèle).

Faire face à la complexité des couches de bas niveau : L'un des principaux challenges lors du développement d'applications en P2P est la nécessité de comprendre les protocoles de bas niveau. Bien que ces protocoles utilisent différents procédés (structures de données et algorithmes) leur objectif sous-jacent reste le même, celui de trouver une donnée particulière dans un réseau P2P, d'une manière évolutive et efficace. Souvent, d'autres exigences non fonctionnelles sont également prises en compte par ces protocoles. Par conséquent, leurs complexités augmentent. Cela les rend difficiles à comprendre et à utiliser par les développeurs d'applications qui ne sont pas forcement des spécialistes de ces protocoles. Les développeurs devraient avoir le choix de construire leurs applications au sein d'une architecture P2P sans faire face à la complexité des couches de bas niveau. Nous avons donc la conviction que l'abstraction est un début de solution pour répondre à ce besoin. En fait, les détails du bas niveau doivent être présentés d'une façon plus abstraite à travers un modèle de haut niveau, clair et facile à comprendre.

1.3.2 DDF : Un langage formel pour des applications P2P à base de composants

Le langage DDF a été développé pour décrire formellement, en utilisant l'approche à composants, des applications P2P et leurs comportements. En particulier, le comportement récursif qui est très fréquent dans le contexte du P2P et que beaucoup d'approches de modélisation ne peuvent pas décrire et analyser (comme c'est expliqué dans la Section 1.2.1). DDF a été également développé pour construire une représentation abstraite des applications spécifiées (c.-à-d., le graphe de dépendance de données). Cette abstraction expose le bon niveau de détails pour effectuer des analyses de flot de données. Avec DDF, une application P2P est spécifiée comme

un réseau de recouvrement (de l'anglais Overlay Network) entre pairs. Les pairs sont représentés par des instances de composants. Chaque instance de composant agit à la fois en tant que serveur (avec ses services d'entrée) et en tant que client (avec ses services de sortie). Chaque instance est connectée à un nombre limité d'autres instances. Quand le réseau évolue dans le temps, les instances peuvent continuellement chercher de nouveaux partenaires en utilisant un protocole comme le protocole Gossip (voir Section 5.2). Dans le cas de DDF, nous supposons l'existence d'une infrastructure sous-jacente (comme le middleware SON) qui fournit aux instances de composants les mécanismes nécessaires de communication et de stockage. Cela nous évite de surcharger la spécification de haut niveau avec les détails liés à la spécificité du réseau, et de traiter ces détails au niveau des couches basses, là où se trouve la nécessité de le faire. Ainsi, nous proposons une spécification simple qui peut être implémentée dans des environnements différents et dynamiques.

1.3.3 Analyse des spécifications DDF en explorant le flot de données

La première étape de cette analyse consiste à construire un graphe de dépendance de données (DDG) à partir de la spécification DDF. Ensuite, la vérification d'une propriété revient à trouver une solution à un ensemble de contraintes (appelées équations de flot de données) sur les entrées et les sorties des nœuds du graphe. Dans le Chapitre 6, nous illustrons cela à travers deux exemples. Le premier exemple consiste à vérifier si un système comporte un interblocage (deadlock), ce qui revient à chercher si un nœud dépend de lui-même dans le graphe. Le deuxième exemple concerne la propriété de dominance (entre données) qui a de nombreuses applications en informatique (optimisation de codes, détection de parallélismes, etc.). Pour calculer la propriété de dominance dans un graphe DDG, nous formulons le problème comme un ensemble d'équations de flot de données qui définissent un ensemble de dominateurs pour chaque nœud du graphe. Ces équations sont résolues grâce à un algorithme itératif.

1.3.4 SON : Un middleware orienté composants, services et P2P

Avec le middleware SON, l'utilisateur est capable non seulement de développer des applications avec une approche à base de composants et orientée services, mais aussi de profiter d'un

mécanisme de génération automatique de code, qui réduit et simplifie plusieurs tâches liées à l'exécution en environnement P2P (comme la gestion de la communication, l'instanciation de composants à distance, la découverte de services, etc.). Ainsi, les développeurs d'applications sont assistés et peuvent se concentrer davantage sur la logique métier. En fait, l'utilisateur de SON définit pour chaque composant un ensemble de services (d'entrée, de sortie et internes). Puis, il implémente seulement le code correspondant, c.à.d. les méthodes associées aux services définis. Ensuite, l'outil de génération de code génère le conteneur de composants qui incorpore toutes les ressources nécessaires pour adapter le code implémenté à l'environnement d'exécution P2P.

1.3.5 Evaluation de SON dans le contexte du projet STAMP

STAMP (modelling dynamic landscapes with **S**patial, **T**emporal **A**nd **M**ulti-scale **P**rimitives) est un projet de recherche financé (en partie) par l'Agence Nationale de la Recherche (ANR) et coordonné par Danny Lo Seen (du CIRAD, un centre de recherche français qui répond, avec les pays du Sud, aux enjeux internationaux de l'agriculture et du développement). Nos contributions dans le cadre du projet STAMP peuvent être présentées en deux grandes phases. Dans la première phase, nous avons participé à la spécification d'un langage de modélisation pour décrire les paysages et leur dynamique. Ce langage est nommé Ocelet et est le résultat principal de ce projet. Dans la seconde phase, nous avons défini pour Ocelet un environnement d'exécution orienté composants et services en se basant sur le middleware SON. L'évaluation de SON dans ce contexte consiste à implémenter des scénarios d'application issus du domaine de la modélisation de l'environnement et de sa dynamique. L'objectif est de montrer comment SON (en particulier, la disponibilité dynamique de services au cours de l'exécution) est capable d'améliorer et de renforcer l'efficacité de ces applications simulant des dynamiques environnementales.

1.4 Organisation du manuscrit

Ce manuscrit de thèse est organisé comme suit. Dans les chapitres 3 et 4, nous présentons les principaux concepts des approches utilisées et nous discutons l'état de l'art des travaux

connexes. Dans le chapitre 5, nous introduisons le formalisme DDF et nous prenons comme exemple illustratif la spécification du protocole Gossip. Dans le chapitre 6, nous présentons comment les algorithmes d'analyse de flot de données peuvent être utilisés pour vérifier les applications spécifiées avec le formalisme DDF. Nous illustrons cela à travers deux exemples : la détection d'interblocage et l'extraction de la relation de dominance. Dans le chapitre 7, nous décrivons les concepts et le fonctionnement du middleware SON. Nous présentons aussi deux prototypes implémentés avec ce dernier : SGT (Simple Georeferencing Tool) qui est une application simple et légère dédiée à la collecte, le traitement et l'affichage de données géoréférencées, et P2Prec (a social based P2P recommendation system) qui est un système de recommandation social en P2P développé au sein de notre équipe de recherche pour le partage de données à large échelle. Le chapitre 8 a pour but de présenter l'évaluation de SON dans le cadre du projet STAMP. Cette évaluation consiste à implémenter des scénarios d'application issus du domaine de la modélisation de l'environnement et de sa dynamique. Deux scénarios d'application ont été implémentés : Lotka-Volterra qui simule l'évolution d'un modèle proie-prédateur, et Rift Valley Fever (la Fièvre de la Vallée du Rift) qui simule la propagation d'une maladie transmise par des moustiques dans une zone de l'Afrique de l'Ouest. Enfin, le chapitre 9 présente la conclusion et les travaux futurs.

Chapter 2

Introduction – in English

Contents

2.1 Overview

Developing Peer-to-Peer (P2P) applications became increasingly important in software development. Nowadays, a large number of organizations from many different sectors and sizes depend more and more on collaboration between actors (individuals, groups, communities, etc.) to perform their tasks. P2P architecture is the concept of an entity acting at the same time as a server and as a client in P2P networks [Schollmeier, 2001]. This is completely different to Client/Server networks, within which the participating entities can act as a server or

as a client but cannot embrace both capabilities. Therefore, the responsibilities of entities are approximately equal and each entity provides services to each other as peers.

In software systems, especially those that support P2P applications, data are required for achievement of the computing activity and driving the interactions between software entities. Nevertheless, software system design is usually based on computational aspects with data as an afterthought. A data-centric approach provides a different way of viewing and designing applications. It lets us focus on the flow and transformation of data through the software system.

In this context, we have defined a Data-Dependency Graph (DDG). It has been chosen as an abstract representation for P2P applications for the following two reasons. Firstly, it represents only one data-flow model (dictated by the dependence between data) on the execution. Further, DDG exposes the right level of detail—enough to perform Data-Flow Analysis (DFA) [Aho et al., 2006].

In this thesis, we present an approach that combines component- and service-oriented development [Szyperski, 1998] [Huhns and Singh, 2005] with well-understood methods and techniques from the field of Attribute Grammars (AGs) [Paakki, 1995] and Data-Flow Analysis (commonly used in compiler construction) in order to specify, analyse and deploy P2P applications. This approach embodies a component-based service middleware called SON (Shared-data Overlay Network) to develop and execute P2P applications, and a formalism called DDF (Data-Dependency Formalism) to capture the behavior of SON's applications and construct their Data-Dependency Graphs.

SON middleware is used as an execution framework to handle the P2P runtime requirements (e.g., communication mechanisms, message queue management and broadcasting messages) with an automatic code generation. This generation offers greater ease to application developers and allows them to focus only on the business logic.

DDF formalism provides the necessary set of operations to specify and analyze P2P applications. DDF can be considered as a minimal and lightweight formalism for the following two reasons. Firstly, the goal of DDF is to formally construct the dependency graph which exposes the right level of detail to perform data-flow analysis. Secondly, DDF is not intended

to express business code or to be a general-purpose programming language. This is performed according to Domain-Specific Language (DSL) [Mernik et al., 2005] principles. We note that DDF is highly inspired by the main characteristics of the Attributed Grammars because they are able not only to construct similar dependency graph, but also to naturally capture complex recursive behavior (which is very frequent in P2P applications cf. Section 2.2.1) that many other approaches cannot describe.

The runtime architecture of SON can be viewed as a set of interacting components. These interactions are performed by receiving or sending service calls. When a service call is received or sent, data can be exchanged (e.g., service parameters, service result and component attributes). Moreover, the propagation of service calls from one component to another may depend on the data carried by a certain service called earlier. Therefore, we want to extend the specification of services that defines the inputs and outputs of components by the notion of dependency. This notion captures not only the dependencies between services, but also the dependencies between exchanged data (required and provided). By defining such notion, it will be possible for a given composition/assembly of components to infer the data-flow and construct a Data-Dependency Graph of the whole system. This notion of dependency between services and between data is defined using DDF.

Once the Data-Dependency Graph is constructed from DDF specification, we can perform several data-flow analyzes. In Chapter 6, we illustrate that through two examples. The first one shows how to treat the deadlock detection problem by searching for circularity in the graph, while the second one computes dominance information by searching for dominators for each graph node. Other analyzes (inspired from DFA and AGs literature (cf. e.g., [Parigot et al., 1996] [Jourdan and Parigot, 1990]) can be performed. For instance, by analyzing the order of data evaluation, we will be able to determine formally which services in a system can be executed in a parallel or incremental way.

In addition to the construction of the dependency graph of a system, our DDF formalism is able to naturally capture recursive behavior by using a rule-based specification. It is a well-known result from the formal language theory that Finite-State Automaton (FSA) cannot capture such behavior [Aho et al., 2006]. This implies that FSA-based approaches used

to model software applications cannot describe and analyze it. In particular, in the context of component-based development, a large body of component behavior modeling approaches can be reduced to FSA. The well-known component models SOFA [Bures et al., 2008] and Fractal [Bulej et al., 2008] clearly raise this issue. For instance, in [Bures et al., 2008] the authors say: *"our approach cannot treat behavior that cannot be modeled by a regular language (e.g. recursion)"*. Therefore, such component approaches are not adequate for P2P applications where recursive behavior is very frequent as explained in the next section.

2.2 Motivations and problem statements

2.2.1 Specificity of P2P applications

Important properties of P2P applications are scalability and self-organization because of their very large user base and the specificity of connections between different peers (e.g., low-bandwidth connections) [Ripeanu et al., 2002]. To support scalability and self-organization in such networks, a large number of P2P-specific algorithms and protocols have been developed. These algorithms and protocols are often executed recursively. Consider, for instance, reputation computation[1] which is a problem of great importance in P2P environments [Aberer and Despotovic, 2001] (a simple example justifying this importance is the case where, while downloading files with a P2P file sharing software, we want to choose only reliable peers). The reputation computation is based on a sequence of queries for getting the trust information about a peer A and their corresponding responses. Such computation should be performed recursively since a response returned from another peer B is the result of a query about the truthfulness of B. In addition, during this trust computation, we must receive all information in the correct order since the cut-off might rely on that order. Such recursive call-backs can be viewed as a sequence of well-formed parentheses if a query call is replaced by a left parenthesis and the corresponding response by a right parenthesis. Therefore, the set of sequences describing these recursive call-backs is a Dyck-Language[2]. It is a well-known result from the

[1] We note that reputation computation presents a particular case of information dissemination and can be performed using Gossip protocol presented in Section 2.2.4.

[2] The Dyck-Language D is the subset of $\{x, y\}^*$ such that if x is replaced by a left parenthesis and y by a right parenthesis, then we obtain sequence of properly nested parentheses [Stanley, 2001].

formal language theory that a Dyck-Language is not a regular language [Stanley, 2001]. Thus, no Finite-State Automaton exists that accepts a Dyck-Language.

The kind of recursive call-backs presented above, which has a properly nested structure, can be well defined in terms of Pushdown Automata or context-free languages (discussed in Section 4.1). However, it is frequently the case that P2P protocols present more complex recursive call-backs which give rise to context-sensitive structures (interactive structures that adjust their behavior when the context changes). Consider, for example, the case where four neighboring peers exchange information according to an interaction that corresponds to two interleaved recursive call-backs. Such kind of interaction ($a^n b^m c^n d^m$) is context-sensitive and cannot be described by context-free languages [Aho et al., 1986].

Referring to the research work on Attribute Grammars [Parigot et al., 1996] which are context-sensitive languages, the recursive behavior of P2P applications can be captured by describing both control and data flow of each interaction. In addition, this behavior can be analyzed using DFA techniques. Furthermore, the Chomsky hierarchy of languages [Chomsky, 1956] ensures the following strict inclusions:

$$type\text{-}3 \subsetneq type\text{-}2 \subsetneq type\text{-}1 \subsetneq type\text{-}0$$

with:

type-3: Regular languages (recognized by Finite-State Automaton).

type-2: Context-free languages (recognized by Pushdown Automaton).

type-1: Context-sensitive languages (recognized by non-deterministic Turing machine).

type-0: Recursively enumerable languages (recognized by Turing machine).

2.2.2 Towards Data-Flow Analysis of P2P applications

2.2.2.1 Model checking and the specificity of P2P applications

Model checking is an automated technique that, given a finite-state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model

[Baier and Katoen, 2008]. It explores all possible states of the system in an exhaustive manner. Model checking has been successfully applied to a wide range of systems such as embedded systems, hardware design and software engineering. Unfortunately, not all systems can take advantage of its power. One reason for this is that some systems cannot be described as a finite-state model. In particular, in the context of P2P applications (as explained above). Another reason is that model checking is not suited for data-intensive applications (which, in many cases, are developed using the P2P paradigm cf. e.g., [Lee et al., 2007] [Ranganathan et al., 2002]). The recent book on model checking [Baier and Katoen, 2008] clearly shows why the verification of data-intensive applications is extremely hard. Even if there are only a small number of data, the state space that must be analyzed may be very large. The authors even consider that this is one of the first weaknesses:

"The weaknesses of model checking:

- *It is mainly appropriate to control-intensive applications and less suited for data-intensive applications as data typically ranges over infinite domains.*
- *Its applicability is subject to decidability issues; for infinite-state systems, or reasoning about abstract data types (which requires undecidable or semi-decidable logics), model checking is in general not effectively computable.*
- *... "*

2.2.2.2 Verification by Data-Flow Analysis

Data-flow analysis refers to a body of techniques, which derive information about the flow of data along software system execution paths [Aho et al., 2006]. The execution of a system can be viewed as a series of transformations of the system state, which consists of the values of all the data in the system. Each execution of an intermediate statement transforms an input state to an output state. We denote these data-flow values before and after a statement s by $INPUTS[s]$ and $OUTPUTS[s]$, respectively.

To analyze the behavior of a system, we must consider all the possible paths (i.e., sequences of system states) through a flow graph that the system execution can take. Thus, solving a problem in data-flow analysis is reduced to find a solution to a set of constraints (called Data-

Flow Equations) on the $INPUTS[s]$ and $OUTPUTS[s]$, for all system statements s. There exist two sets of constraints:

- Semantic constraints: they define the relationship between $INPUTS[s]$ and $OUTPUTS[s]$ of each statement s. This relationship is usually presented as a transfer method f that takes the $INPUTS[s]$ before the statement and produces $OUTPUTS[s]$ after the statement. That is, $OUTPUTS[s] = f_s(INPUTS[s])$.
- Control-flow constraints: If a system consists of statements $s_1, s_2, ..., s_n$, in that order, therefore, the control-flow value out of s_i is the same as the one into s_{i+1}. That is, $INPUTS[s_{i+1}] = OUTPUTS[s_i]$, for all $i = 1, 2, ..., n - 1$.

For example, to verify a property such as liveness of data that determines whether a datum is used in the future along some path in the flow graph, we shall set up the constraints for liveness of data (i.e., define the data-flow equations specifying that a datum d is live at a system point p if some path from p to its end contains a use of d). These equations can be solved using an iterative algorithm form a fixed-point solution. The convergence of the algorithm is assured by the theory of iterative data-flow analysis [Kam and Ullman, 1976], which demonstrates that a unique fixed point exists for these equations. Liveness information can be very useful. For instance, if the result of a datum assignment in a software system is not used along any subsequent execution path, then the assignment is considered as dead code that we can eliminate. In Section 6.3.2, we provide an other example (detection of dominance) that illustrates in more details the principles of data-flow analysis.

A broad range of other system properties can be computed at this level of data abstraction, including some properties like safety and liveness that model checking cannot compute for infinite state systems (cf. e.g., [Govindarajan et al., 1992]). In addition, several algorithms have been proposed in literature to compute these properties. Unfortunately, to date, the most dominant application of these algorithms, and more generally, Data-Flow Analysis, is in the context of compiler construction. In particular, for Attribute Grammar formalism, which is used to describe the semantic analysis in most compilers.

Our motivation in this context is to use the well-understood methods and techniques from the field of AGs and DFA in order to construct an abstract representation for P2P applications

and then perform data-flow analyzes on it.

2.2.3 Exploring data-centric approach for component-based systems

Component-based Software Engineering (CBSE) [Szyperski, 1998] became increasingly important in software engineering. This emerges from the need to use CBSE concepts to implement services and raise the level of abstraction by easing packaging, reusing, extending, customizing and composing services [Yang and Papazoglou, 2004]. Thus, services can be encapsulated and their interfaces can be exposed into cohesive components to assist in the creation of new applications. Hence, component-based approach yields promising benefits such as service composition, reusability and adaptation. However, the data manipulated by services to produce actionable results and which drive component interactions are considered as an afterthought. Whereas, the data are incorporated as an important part of the development of systems in several research areas such as Grid Computing, Business Intelligence and P2P systems. Recently, in the emerging Cloud Computing area, where everything is as a service, data management has been receiving significant excitement and attention [Abadi, 2009], and this can only increase.

Our motivation in this context is to investigate the applicability of the data management for software component systems by allowing run-time data to be specified, viewed and analyzed, especially in P2P environments. While many of the current component approaches emphasize the structural and functional aspects of component composition, we insist on modeling of flow and dependencies of run-time data because the interactions between components are due to exchanged data. Thus, it is our belief that data must be considered to be an integral part of design and behavior specifications of component-based systems.

2.2.4 Illustrative example: specifying Gossip protocol

In order to motivate and illustrate that our approach is useful, especially in the context of P2P applications, we explain our dependency formalism in an example that consists of a Gossip protocol [Voulgaris et al., 2005, Jelasity et al., 2007]. Gossip protocol, also called epidemic protocol, is well-known in the community of P2P. It is mainly used to ensure a reliable information dissemination in a distributed system in a manner closely similar to the spread of

epidemics in a biological community. This kind of dissemination is a common behavior of various P2P applications, and according to [Jelasity, 2011], a large number of distributed protocols can be reduced to Gossip protocol. There exist different variants of Gossip protocol. However, a template that covers a considerable number of those variants has been presented by Jelasity in [Jelasity, 2011]. In our example, we will rely on this template shown in Algorithm 2.

Algorithm 2 The gossip algorithm skeleton *(from [Jelasity, 2011])*

```
loop
   timeout(T)
   node ← selectNode()
   send gossip(state) to node
end
procedure onPushAnswer(msg)
   send answer(state) to msg.sender
   state ← update(state, msg.state)
end
procedure onPullAnswer(msg)
   state ← update(state, msg.state)
end
```

To model this Gossip protocol, we consider a set of nodes, which get activated in each T time units exactly once and then spread data in a network by exchanging messages. Basically, when a node receives data, it responds to the sender and propagates the data to another node in the network (in practice, the data are propagated to a subset of nodes selected according to a specific algorithm). In terms of service, a *node* is a component that has two activities: serving and consuming data. There are two input services for the serving activity and two output services for the consuming activity. These services are described in the *node interface* as follows:

$$(\{answer(resp : String), gossip(info : String)\}_{in},$$
$$\{gossip(info : String), answer(resp : String)\}_{out})$$

The *gossip* service is for the propagation of data and the *answer* service is for sending a response to the sender. The behavior of input services (serving activity) just mirrors the same steps of the output services (consuming activity). From this description of services, we can construct intuitively a simple dependency graph between services, i.e., output services of a $node_x$ are connected to input services of $node_y$, and so on. This graph represents a part of the control flow but it is not very explicit about the data flow. In fact, we do not know the

dependencies between services and between data within a *node*.

To complete this interface with a description of both control and data flow, our formalism specifies the behavior with a set of rules:

$$
\begin{array}{lll}
r_1: & timeout(T) & \rightarrow (gossip(state_x), node_y) \\
r_2: & (gossip(state_y), node_y), [onPush] & \rightarrow (answer(state_x), node_y) \\
r_3: & (gossip(state_y), node_y), [onPull] & \rightarrow \\
r_4: & (answer(state_y), node_y) & \rightarrow
\end{array}
$$

where, r_1 indicates that the internal service *timeout* activates the $node_x$ in each T time and then sends the data $state_x$ to $node_y$ through the service *gossip*. r_2 indicates that the $node_x$ receives the data $state_y$ from $node_y$ and then responses by sending the data $state_x$ through the service *answer* if the condition *onPush* is satisfied. *onPush* is a guard condition (to keep things simple, we will ignore guard conditions in this example). r_3 indicates that the $node_x$ receives the data $state_y$ from $node_y$ through the service *gossip*. r_4 indicates that the $node_x$ receives the data $state_y$ from $node_y$ through the service *answer*.

By introducing these rules, the system can be viewed as a set of components where each component has inputs (left side of the rules) and outputs (right side of the rules). The inputs receive data carried by services, and after computation, these data can be sent through outputs. Therefore, we can extract a Data-Dependency Graph of the whole system by connecting together the partial data dependency graphs corresponding to each component used in this system.

2.2.5 Needs for component and service-oriented P2P runtime

There exist interesting technologies developed in the P2P context. However, most of these technologies are not well exploited in application development process due to the limitations of specification approaches used for P2P applications. One of these limitations is the tight coupling between application specification and the underlying P2P technologies and protocols. This forces software developers to make tedious efforts in finding and understanding detailed knowledge about P2P low level concerns. Moreover, this tight coupling constraints applications to run in a changeless runtime environment. Consequently, choosing (for example) another

protocol at runtime to meet a new requirement becomes very difficult. Besides these previous issues, P2P applications are usually specified with a weak ability to delegate computing activities between peers, and especially focus on data sharing and storage. Thus, it is not able to take full advantages of the computing power of the underlying P2P network.

Service-Oriented Architecture (SOA) is an approach for designing and architecting loosely coupled applications with services requested and consumed on demand. The main purpose of SOA is to provide a flexible model of integration (by reducing dependencies) as well as a higher level of abstraction (through encapsulations of details). Thus, the ability to align applications with new business requirements increases. In the literature, SOA are usually coupled with CBSE principles to propose component-based service middlewares. However, most of them are not adapted to P2P applications. One reason for this is that they rely on centralized service registries/brokers. Such centralized element in a SOA might cause a bottleneck and central point of failure. Thing that introduces reliability risks and limits application scalability. Another reason is that they are based on communication protocols which are inadequate in P2P environment. For instance, P2P systems are not forced to operate using a Domain Name Service (DNS) because the peers might not have a permanent IP address.

In this thesis, we present SON middleware. It extends the principles of service-oriented architecture as well as component-based development to support building applications within a P2P architecture in an effortless and effective way.

SON middleware assists application developers by providing an automatic code generation which handles several runtime requirements (e.g., communication mechanisms, message queue management, broadcasting messages, etc.). In fact, SON's user implements only the business code corresponding to the declared services. Afterwards, a code generation tool generates the corresponding components and their associated containers. The component container embodies all resources needed to adapt the implementation code to the P2P runtime environment.

SON can be considered as a generic lightweight P2P middleware (with the necessary set of operations that must be present to develop component and service-based P2P applications) for the following reason. Since, in most cases, the challenges of P2P systems can be reduced to a single problem: *"How do you find any given data item in a large P2P system in*

a scalable manner, without any centralized servers or hierarchy?" [Balakrishnan et al., 2003], SON has been unified the notion of publish/subscribe: it uses a DHT (Distributed Hash Table) [Rhea et al., 2004] not only to publish and subscribe data, but also to enable dynamic service publication, discovery, and deployment.

2.3 Contributions

The work of this thesis features four major parts of contributions. Before giving a summary of each part in the last four subsections, we present first key ideas that underlie those parts.

2.3.1 Key ideas in our contributions

Pay attention to dependencies: Any software specification that is expressed in a language contains some kinds of dependencies between data and between the steps of the specification. Software developers generally pay little attention to these dependencies, in particular, to "non-direct" dependencies that they can be very hard to identify without a computer analysis. In many cases, managing dependencies leads to direct improvement in the application's running time. For example, reducing the number of dependencies may help to perform several optimizations (e.g., in execution time or memory usage).

Separate what is computed from how it is computed: Roughly speaking, there are three times at which dependencies can be detected and adjusted: when software is specified, when it is compiled, and when it is executed. In our case, we are interested in dependencies at the specification level and we have chosen to separate, as far as possible, what is computed from how it is computed. The advantage of this choice comes from the fact that not only one but multiple implementations of the specification can be synthesized by analyzing data dependencies (e.g., evaluating data in incremental, partial, or parallel way).

Avoid the stress of dealing with low level complexity One of the main challenging issues in P2P application development is the need to understand low level protocols. Although low level protocols use various schemes, data structures and algorithms, the underlying purpose remains the same: find given data within a P2P network in a scalable and consistent

manner. In many cases, other non-functional requirements are also taken in charge by the protocols. Consequently, their complexity increases, which makes them difficult to comprehend and use by non-specialist software developers. Software developers should have the choice to build their applications within a P2P architecture without the stress of dealing with P2P low level complexity. As such, we believe that abstraction is a solution to simplify the development of P2P applications. The low level details should be abstracted into clear and easy to understand model.

2.3.2 DDF: A formal language for component-based P2P applications

DDF (Data Dependency Formalism) is used as an underlying formalism for the work presented in this thesis. It has been developed to formally describe component-based P2P applications and their recursive behavior. This kind of behavior is very frequent in the context of P2P and many modeling approaches cannot describe it, as explained in Section 2.2.1. DDF has been also developed to construct an abstract representation (i.e., Data-Dependency Graph). This abstraction exposes the right level of detail to perform data-flow analyzes. With DDF, a P2P application is specified as an overlay network between peers. Peers are represented by component instances. Each component instance acts both as a server (with its input services) and a client (with its output services). Each instance is connected to a bounded number of other instances. As the network evolves, instances can continuously seek after new partners through a specific protocol, such as Gossiping (cf. Section 5.2). We assume the existence of an underlying layer (SON infrastructure in our case) that provides to component instances the necessary storage and communication mechanisms. These assumptions allow us to make only very weak networking issues at the high level description and defer the additional ones to the lowest level where they are needed. Thus, we provide a simple specification that can be implemented in different environments with different low level assumptions.

2.3.3 Analysis of DDF specification with data-flow principles

The first step of this analysis is to construct a Data-Dependency Graph (DDG) from a DDF specification. After that, verifying a property is reduced to find a solution to a set of constraints

(called data-flow equations) on the inputs and the outputs of the graph nodes representing data. In this thesis, we illustrate that through two examples. The first example consists of checking the property of deadlock freedom, which is reduced to find whether a node in the graph depends on itself. The second example is about dominance property that has many applications in computer science (code optimization, detection of parallelism, construct of hierarchical overlay networks, etc.). To compute dominance information in a DDG, we formulate the problem as a set of data-flow equations that defines a set of dominators for each graph node. These equations are solved with an iterative algorithm.

2.3.4 SON: A component- and service-oriented P2P middleware

By using SON (Shared-data Overlay Network) middleware, the user is able not only to execute applications in component- and service-oriented model, but also to perform an effective code generation to support P2P runtime requirement. Thus, software developers are assisted and have greater ease in application development stage. These facilities allow them to focus more on the business logic and defer to SON the management of the runtime (e.g., communication mechanisms, instantiation and connection of components, service discovery, etc.). In fact, SON's user defines for each component a set of services (input, internal and output). Then, he only implements the code of the component, i.e., the methods that implement the defined services. Afterwards, a code generation tool, called Component Generator, generates the component container that embodies all resources needed to adapt the implementation code to the P2P runtime environment.

2.3.5 Evaluation of SON in the STAMP project

STAMP (modelling dynamic landscapes with **S**patial, **T**emporal **A**nd **M**ulti-scale **P**rimitives) is a research project supported (in part) by the Agence Nationale de la Recherche (ANR) and coordinated by Danny Lo Seen (from CIRAD, a French research centre working with developing countries to tackle international agricultural and development issues). Within the STAMP project, we have contributed in two main ways. First, we have participated to the design and the specification of an environmental modelling language called Ocelet, which has been the main

result of this project. Second, we have defined for Ocelet a component and service-oriented runtime based on SON infrastructure. The evaluation of SON in this context consists of implementing application scenarios from the area of modelling environmental landscapes and their dynamics. The objective is to show, how SON (i.e., especially the dynamic availability of services in a service-oriented runtime) is able to improve and enhance the effectiveness of such environmental dynamic applications.

2.4 Thesis outline

This thesis is organized as follows. In Chapters 3 and 4, we present some background concepts and the state-of-the-art. In Chapter 5, the DDF formalism is introduced and illustrated through the case-study Gossip protocol. In Chapter 6, we present how Data-Flow Analysis techniques can be used to analyze applications specified with our DDF. We illustrated that through two examples: deadlock and dominance detection. In Chapter 7, we describe the fundamental concepts of SON middleware. Besides the conceptual issues, the chapter presents a summary of the two prototypes: SGT (Simple Georeferencing Tool) which is a lightweight application dedicated to collect, process and display georeferenced data, and P2Prec (a social based P2P recommendation system) which is developed in our research team for large-scale data sharing. Chapter 8 aims at presenting the results of the evaluation of the SON in the context of STAMP project. The evaluation consists of implementing application scenarios from the area of modelling environmental landscapes and their dynamics. Two application scenarios are presented: Lotka-Volterra that simulates the evolution of a predator-prey model, and Rift Valley Fever that presents a land-scape modelling experiment on the spread of a mosquito-borne disease in an arid area in West Africa. Finally, Chapter 9 concludes and presents future work.

Part II:

Background and State-of-the-art

Chapter 3

Paradigms and concepts

Contents

Before engaging in a discussion of existing related approaches (presented in the next chapter), we aim here to familiarize the reader with some concepts from the field of software engineering that are needed to comprehend the work presented in this thesis. These concepts are presented in three sections. Section 3.1 defines the concept of a software component and gives the principles of component-based development. It also presents the main characteristics of some industrial and academic component models. Section 3.2 introduces Service-Oriented Architecture (SOA), and presents its characteristics and its design principles. Finally, Section 3.3 gives an overview of Peer-to-Peer (P2P) architecture and presents some of its related designs (i.e., centralized, decentralized and hybrid).

3.1 Component orientation

3.1.1 What is a component?

In its most general sense, a software component is an independently deliverable package of reusable software services [Kaisler, 2005]. In the computer science literature, the term "component" has many definitions. Here are a few commonly accepted ones:

> - *A component is a non-trivial, nearly-independent, and replaceable part of a system that fulfills a clear function in the context of a well-defined architecture. A component conforms to and provides the physical realization of a set of interfaces.* [Kruchten, 1998]
>
> - *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition.* [Szyperski, 2002]
>
> - *A software component is a piece of self-contained, self-deployable computer code with well-defined functionality and can be assembled with other components through its interface.* [Wang and Qian, 2005]

Kruchten suggests that, first, a component is non-trivial; it is functionally and conceptually larger than a single class or a single line of code. Typically, a component encompasses the structure and behavior of a collaboration of classes. Second, a component is nearly independent of other components because it rarely stands alone. A given component collaborates with other components and in so doing assumes a specific architectural context. Third, a component is substitutable for any other component which realizes the same interfaces. This aspect helps during development, where parts of a system can be stubbed, sketched, then replaced by mature, robust implementations. Fourth, a component fulfills a clear function. A component is logically and physically cohesive, and thus denotes a meaningful structural and/or behavioral chunk of a larger system. It is not just some arbitrary grouping. Fifth, a component exists in the context of a well-defined architecture. A component represents a fundamental building block upon which systems can be designed and composed. Finally, a component conforms to a set of interfaces. Thus, it may be substituted in any context wherein that interfaces apply.

Szyperski suggests[1] that a component has a technical part, with aspects such as independence, contractual interfaces, and composition. It also has a market-related part, with aspects such as third parties and deployment. This definition has several implications. For a component to be composable with other components, it needs to be sufficiently self-contained. Also, it needs to come with clear specifications of what it requires and provides. In other words, a component needs to encapsulate its implementation and interact with its environment by means of well-defined interfaces. For a component to be independently deployable, it needs to be well separated from its environment and other components. A component, therefore, encapsulates its constituent features. Also, as it is a unit of deployment, a component will never be deployed partially. In this context, a third party is one that cannot be expected to have access to the construction details of all the components involved.

Wang and Qian suggest that a component is a program or a collection of programs that can be compiled and made executable. It is self-contained; thus, it provides coherent functionality. It is self-deployable so that it can be installed and executed in an end user's environment. It can be assembled with other components so that it can be reused as a unit in various contexts. The integration is through a component's interface, which means that the internal implementation of a component is usually hidden from the user. This definition differs from the previous ones by the fact that it has been proposed to embrace a wide range of industrial component-based approaches. Component approaches complying with it include JavaBeans and Enterprise Java Beans (EJB) from Sun Microsystems (bought by Oracle), COM (Component Object Model), DCOM (Distributed Component Object Model), and .NET components from Microsoft, and CORBA (Common Object Request Broker Architecture) components from the Object Management Group.

3.1.2 Component-Based Software Development (CBSD)

Component-Based Software Development (CBSD) is also called component-based software engineering (CBSE). Its main purpose is to break monolithic applications into reusable units

[1] Szyperski's definition was first formulated at the 1996 European Conference on Object-Oriented Programming (ECOOP) as one outcome of the Workshop on Component-Oriented Programming (Szyperski and Pfister, 1997).

(components) that can be implemented, distributed, and upgraded independently [Kaisler, 2005]. To achieve this, we need mechanisms for interoperability between components. Once components can interoperate, we can combine them to develop larger and more complex applications in an incremental way. Thus, CBSD improves software developers' productivity and application quality.

As suggested in [Leeb, 1996], component-based development has at least two key advantages over traditional Object-Oriented Programming (OOP). First, components interoperate at runtime. Therefore, they can be dynamically integrated in an application. Second, component interfaces are separated from the implementation. Thus, it is possible to implement and update components independently. To implement the code of a component, only the interface of that component is needed.

At the implementation level, applications are generally sets of modules written in one or more programming languages. Such modules come under various names (methods, procedures, objects, packages, etc.), but they can all be seen as abstractions for components. However, they are not sufficient to meet the goal of component-based development. One reason for this is that the programming languages used to develop these modules only support a small set of basic interconnection mechanisms (e.g., method invocation or shared global variables). The programmer is then constrained to include additional functionalities to reduce dependencies among application modules.

3.1.3 Component models

A component model consists of a set of rules to be followed in component development and deployment [Jaffar-ur Rehman et al., 2007]. These rules might concern the way in which the interfaces should be implemented, the component that must be packaged, and the documents to be filled out to provide additional information on the component itself. Many different component models have been defined and they can be put in two large groups: industrial group and academic group. In the following sections, we briefly present three component models that lead the scene in each group. More discussions about other academic ones related to our approach can be found in the next chapter.

3.1.3.1 Academic models

SOFA 2.0 [Bures et al., 2006] is a typical academic component model. It uses hierarchical components that can be either primitives or composites. It is a successor of the SOFA component model [Plásil et al., 1998], which has the following features: ADL-based design, behavior specification using behavior protocols, generated connectors supporting distribution of applications, and a runtime environment with dynamic update of components. In SOFA 2.0, a component is primarily seen as a black-box. It has well-defined interfaces and exists at design, deployment, and run time. Components are defined by their frame and architecture. A frame enables to define a component via interfaces, while an architecture implements at least one frame and specifies internal structure of the component (its subcomponents and their composition). The specification is separated from the implementation by using meta-model. The semantics of component composition is defined trough Extended Behavior Protocols (EPB). Finally, deployment-related features are specified separately in a deployment plan.

FRACTAL [Bruneton et al., 2006] is a general component model that is dedicated to implement, deploy, and manage software systems, in particular, operating systems and middleware. The main features of FRACTAL are the following: composite components (that contain sub-components), in order to have various levels of abstraction; shared components, in order to model and share resources while maintaining component encapsulation; Introspection capabilities, in order to control the execution of a system; and re-configuration capabilities, in order to dynamically deploy and configure a system. A FRACTAL component can be understood as being composed of a membrane that supports interfaces to introspect and reconfigure its internal features, and a content that consists of a set of other components. The membrane of a component can have external and internal interfaces and is typically composed of several controllers (that usually play the role of interceptors). In addition to component nesting, FRACTAL provides bindings mechanisms to define the architecture of an application. Communication between components is only possible if their interfaces are bound. FRACTAL supports two kinds of binding: primitive and composite. A primitive binding is a binding between a client interface and a server interface in the same address space (i.e., it can be implemented by direct

language references). A composite binding is a communication path between a certain number of component interfaces. This allows to construct a distributed configuration of FRACTAL components.

JAVA/A [Baumeister et al., 2006] is a programming language that forms an instance for a Java-based architectural programming. JAVA/A integrates architectural notions into Java, and provides an abstract component model. In contrast to interface-based component approaches, the primary distinguishing characteristic of JAVA/A component model is the consistent use of ports as explicit architectural modelling elements [Knapp et al., 2008]. Ports allow application designers to segment the communication interface of components and thus the representation of different views (faces) to other components. In addition, ports are equipped with behavioral protocols to regulate message exchange according to the desired viewpoint. Furthermore, the fact to strongly encapsulate behaviors communicating through ports fosters modular verification, which is one of the aims of the JAVA/A approach. Another important aim of the JAVA/A is the representation of software architecture entities in a programming language. JAVA/A then extends Java by introducing keywords: *port*, *required*, *provided*, *simple* and *composite component*, and *assembly*, and including port protocol descriptions as UML state machines. The JAVA/A compiler transforms components into Java code that can be compiled to byte code using a Java compiler. The generated Java classes are integrated into the JAVA/A component framework, which provides operations that are common to all JAVA/A components (like reconfiguration support).

3.1.3.2 Industrial models

Enterprise JavaBeans (EJB) [Panda et al., 2007] is an architecture that defines a programming model for developing server-side Java applications. It provides an EJB container to manage the life cycle of application components. When an EJB client requests a server component, the container allocates a thread and submits the request to the appropriate EJB component instance. The container manages all component resources and all interactions between components and the external systems. The EJB component model defines the structure of the component interfaces and the mechanisms through which a

component interacts with its container and with other components [Gorton, 2011]. The EJB version 1.1 defines two main types of components, called session beans and entity beans. Session beans are generally used for executing business logic and to respond to the clients' requests. In a model-view-controller architecture, session beans correspond to the controller because they embody the business logic of a three-tier architecture. There are two types of session beans, namely stateless session beans and stateful session beans. A stateless session bean does not maintain a conversation with calling process. This means that it does not keep any state information related to any client that calls it. Contrary to a stateless session bean, a stateful session bean maintains a conversation with calling process; therefore it keeps state information about the client that calls it. On the other hand, entity beans are generally used for representing business data. Data objects in an entity bean are mapped to some data items in an associated database. Entity beans are usually accessed through session beans, which provide the business level services to the application clients. For further reading, we refer to the online EJB specification [Oracle, 2012] that is continuously updated.

CORBA Component Model (CCM) [OMG, 2006] [OMG, 2002] is defined by the Object Management Group (OMG), which is an open membership, not-for-profit computer industry standards consortium. CORBA (Common Object Request Broker Architecture) has been proposed to enable software components written in different computer languages and running on different computers to work together. In CORBA, a component is a basic meta-type. The component meta-type is an extension and specialization of an object meta-type. Component types are specified in IDL (Interface Definition Language) and represented in an Interface Repository. A component is denoted by a component reference that is represented by an object reference. A component definition is a specialization and extension of an interface definition. Although the current CORBA specification does not provide mechanisms to support formal semantic descriptions, component definitions are associated with a single well-defined set of behaviors. A component type encapsulates internal representation and implementation, and it is instantiated to create concrete instances with state and identity. Although the specification of components includes standard frameworks for implementation, these frameworks, and any assumptions that they

might involve, are completely hidden from the component clients.

OSGi [The OSGi Alliance, 2012] is a general-purpose Java framework that supports the deployment of extensible and downloadable applications known as bundles. OSGi-compliant devices can download, install or remove OSGi bundles. The management of the installation and the update of bundles is dynamic and scalable in the run time. OSGi framework provides to the bundle developer the necessary resources needed to take advantage of independence and dynamic code-loading capability in order to effortlessly develop services that can be deployed on a large scale in small-memory devices. In OSGi, a component is a Java class contained in a bundle. The distinguishing aspect of a component is that it requires the following artifacts within the bundle: i) an XML document that contains the component description; a Service-Component manifest header that names the XML documents with the component descriptions; and an implementation class which is specified in the component description. Component configurations are activated and deactivated under the control of SCR (Service Component Runtime - the actor that manages the components and their life cycle). The decisions of SCR are based on the information specified in the component's description. This information consists of basic information about the component like the name, type, implemented services and references. References are dependencies which the component has on other services.

3.2 Service-Oriented Architecture (SOA)

3.2.1 Definition and characteristics

W3C Working Group defines Service-Oriented Architecture (SOA) as "*a set of components which can be invoked, and whose interface descriptions can be published and discovered*" [W3C, 2004]. The resources provided by these components are called services and a service is defined as follow: "*a service is an abstract resource that represents a capability of performing tasks that form a coherent functionality from the point of view of providers entities and requesters entities. To be used, a service must be realized by a concrete provider agent*" [W3C, 2004]. As illustrated in Figure 3.1, SOA relies on three actors: i) the Service Provider

publishes on a Service Broker the service descriptions which specify both the available service operations and how to invoke them (e.g., network protocol that must be used for the invocation, software components required to establish the connection, etc.); ii) the Service Broker registers the service descriptions and references; and iii) the Service Consumer discovers the services by running a search on the Service Broker. It then establishes a connection with the provider to invoke the service operations.

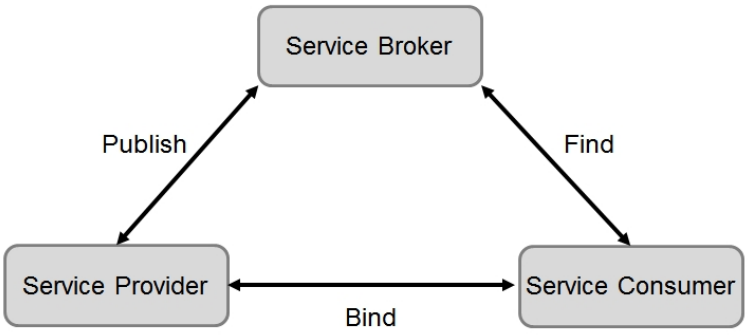

Figure 3.1: Actors in Service-Oriented Architecture.

3.2.2 Design principles

Principles help shape every aspect of our world. We navigate ourselves through various situations and environments, guided by principles we learned from our society and from our own experiences. In the IT world, many approaches encouraged the use of design principles so that when you did something, you would "do it right" on a consistent basis. Often, though, their use was optional or just recommended. They were viewed more as guidelines than standards, providing advice that we could choose to follow. When moving toward a service-oriented architecture, principles take on renewed importance primarily because the stakes are higher. Instead of concentrating on the delivery of individual application environments, we usually have a grand scheme in mind that involves a good part of the enterprise. A "do it right the first time" attitude has therefore never been more appropriate [Erl, 2007].

To achieve this, Erl proposes in its book [Erl, 2007] eight SOA design principles: service contracts, service coupling, service abstraction, service reusability, service autonomy, service statelessness, service discoverability and service composability. In the following paragraphs, we give a brief explanation (extracted from Erl's book) for each of these design principles.

Service contract. As with many terms in the IT industry, "contract" is one that can have different meanings when associated with automation solutions. For example, it is relatively common to view a contract as the equivalent of a technical interface. When it comes to services within SOA, we have a slightly broader definition. A contract for a service (or a service contract) establishes the terms of engagement, providing technical constraints and requirements as well as any semantic information the service owner wishes to make public. A service contract is always comprised of one or more technical service descriptions designed for runtime consumption, but there are also cases when non-technical documents are required to supplement the technical details. Both are considered valid parts of the overall contract.

Loose coupling. Any part of an automation environment that's separable has the potential (and usually the need) to be coupled to something else for the sake of imparting its value. The root of the term (couple) itself implies that two of something exist and have a relationship. The most common way of explaining coupling is to compare it to dependency. A measure of coupling between two things is equivalent to the level of dependency that exists between them. In SOA, we emphasize the reduction ("loosening") of dependencies ("coupling") between the parts of a service-oriented solution, especially when compared to how applications have traditionally been designed. Specifically, loose coupling in SOA is advocated between a service contract and its consumers and between a service contract and its underlying implementation.

Service abstraction. We can only assess and judge the value of something for which information is made available to us. What we publish about a service communicates its purpose and capabilities and provides details to potential consumers about how it can be programmatically invoked and engaged. The information we don't publish about a service protects the integrity of the coupling formed between it and its future consumers. By keeping specific details hidden, we allow the service logic and its implementation to evolve over time while continuing to fulfill its obligations in relation to what was originally published in its contract. Service abstraction therefore raises post-implementation, organizational issues (such as access control) that can also be part of a governance methodology. However, because it directly affects the service design process and specifically influences

design-time decision points as to what should be published in the official service contract, it is very much part of the service design stage as well.

Service reusability. There is perhaps no principle more fundamental to achieving the goals of service-oriented computing than that of reusability. It can even be argued that several of the other principles would not exist if the service-orientation paradigm did not place such a core emphasis on fostering reuse. In theory, reuse is a pretty straight-forward idea: simply make a software program useful for more than just one single purpose. The reasons for doing so are also quite evident. Whereas something that is useful for a single purpose will provide value, something that is repeatedly useful will provide repeated value and is therefore a more attractive investment. The rationale is logical, but it also brings to light the difference between something that is simple and something that is easy. Reuse is a simple concept, but history has taught us that achieving reuse is not easy.

Service autonomy. Autonomy represents the ability to self-govern. Something that is autonomous has the freedom and control to make its own decisions without the need for external approval or involvement. Therefore, the level to which something is autonomous represents the extent to which it is able to act independently. If a software program exists in an autonomous runtime state, it is capable of carrying out its logic independently from outside influences. It therefore must have the control to govern itself at runtime. The more control the program has over its runtime execution environment, the more autonomy it can claim. Hence, for services to provide a consistently reliable and predictable level of performance as members of a service inventory and as members of complex compositions, they must exist as self-sufficient parts, i.e., possess a significant degree of control over their underlying resources.

Service statelessness. A good indication that the design of an agnostic service was successful is when it is reused and recomposed on a regular basis. This outcome emphasizes the need to optimize the service processing logic so as to support the requirements of multiple consumer programs while the service itself consumes as little resources as possible. As the complexity of service compositions increases, so does the quantity of activity-specific data that needs to be managed and retained throughout the lifespan of the composition.

Services required to process and hold this data while waiting for other services in the composition to carry out their logic can tax the overall infrastructure. This is especially the case when numerous instances of those services need to exist concurrently, further compounding the drain on system resources. To maximize service scalability and to make the most of whatever performance thresholds service inventories are required to work within, services and their surrounding architecture can be designed to support the delegation and deferral of state management responsibilities. This result in a service design streamlined by leveraging a condition called statelessness.

Service discoverability. The concepts behind discovery are quite straight-forward. From an architectural perspective, it is often desirable for individual units of solution logic to be easily located. The process of searching for and finding solution logic within a specified environment is referred to as discovery. A key aspect of discovery is that you may or may not have been aware of the logic's existence before you discovered it. By discovering that something you want to build already exists, you avoid creating redundant logic. By discovering that something you want to build does not yet exist, you can safely define the scope of your development effort. Discovery is often classified as an extension of infrastructure and therefore associated with application architecture. For something within the application to be discoverable, it needs to be equipped with meta-information that will allow it to be included within the scope of discovery searches. An architectural component that can adequately be discovered is considered to have a measure of discoverability. In term of service, discoverability information is essentially a combination of the content in a service contract and meta-data in the corresponding service broker.

Service composability. If something is decomposed, it can be recomposed. In fact, composition is usually the reason something is decomposed in the first place. We break a larger thing apart because we see potential benefit in being able to do things with the individual pieces that we would not have the freedom to do were they to exist as just a whole. Applying this approach establishes an environment where solution logic exists as composable units. As a result, there is the constant opportunity to recompose the same solution logic in order to solve new problems. When we apply this rationale to the world of automation, the implications become pretty clear. Why build one large program that can only perform

a fixed set of functions, when we can decompose that program into smaller programs that can be combined in creative ways to provide a variety of functions for different purposes? This is the basis of the separation of concerns theory supported in SOA through service composition principle.

3.3 Peer-to-Peer (P2P) architecture

3.3.1 What is Peer-to-Peer?

Originally, the term peer-to-peer was used to describe the communication between two peers and is analogous to a telephone conversation. A conversation through a telephone involves two people (peers) which have equal status, communication between a point-to-point connection. Simply, this is what P2P is, a point-to-point connection between two equal participants [Taylor and Harrison, 2004].

Historically, the Internet started as instances of P2P systems. Its challenge was to establish connections among distributed machines using different network protocols and within a common network architecture. The first hosts on this network were some US universities, which had independent computing sites connected with equal status, not in client/server or master/slave relationship. From the beginning of the Internet until mid-nighties, internet network had one model of connectivity. This model assumed that machines are always switched on and connected with permanent IP addresses. However, with the development of the first global web browser called Mosaic and the invention of the dial-up modem, a new model of connectivity began to emerge because users would enter and leave the Internet frequently and unpredictably. Therefore, the DNS system has been evolved to support assigning IP addresses dynamically.

Over time though and the improvement of software and hardware technologies, P2P has begun to emerge as a class of applications that takes advantage of the second internet connectivity model. This class of applications started by exploiting unused resources in the network such as storage space, communication edges and available processors. Among the first proposed applications, we find Napster [Napster, 2012] launched in 1999, Gnutella [Gnutella, 2012] launched in 2000, and Freenet [Freenet, 2012] launched in 2001. They were especially dedicated to

file sharing—users wanted to find certain files (e.g., music or video files) in the Internet network and download them as soon as possible. Actually, P2P applications are not limited to file sharing and are developed for various concerns like voice over IP, instant messaging, video streaming and anonymous web browsing. Although these P2P applications relay on the same P2P principles, they have been developed in different P2P architecture designs that we describe in the next subsection.

3.3.2 Architecture designs

As presented by the IEEE Standards Association *"architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution"* [IEEE, 2000]. From this definition, architecture captures the structure of a system in terms of components and how they interact. It also notes that architecture has a design and evolution principles. In the P2P context, this corresponds to how the participating peers (components), at the application level, connect among each other, and how they need to fulfill their obligated tasks to maintain an evolving network [Kwok, 2011]. Hence, if we change how peers connect, interact and evolve we may obtain different P2P architectures. Based on existing P2P systems that have been developed, we can classify possible P2P architectures into three categories: centralized, decentralized and hybrid.

3.3.2.1 Centralised P2P systems

Although P2P is usually considered as an opposite of the centralized client-server model, the first generation of P2P systems (e.g., Napster) relayed on centralized architecture. Nevertheless, in contrast to traditional client-server model, the central server(s) in P2P systems only keeps meta-information about shared content (e.g., ID or IP addresses of peers where a content is available) and manages global tasks (e.g., deals with updates in the network and coordinates tasks among the peers) [Liu and Antonopoulos, 2010]. However, as in decentralized systems, once a peer obtains its information and tasks, it can connect and interact directly with other peers (without going through the central server(s) anymore).

Although P2P systems based on centralized architecture are pretty efficient since the interaction among peers is facilitated by central server(s), such systems usually fail to scale with the increase of participating peers. The central server(s) rapidly become the performance bottleneck and the existence of single point of failure prevents from using network for many potential applications [Galuba and Girdzijauskas, 2009].

3.3.2.2 Decentralised P2P systems

Due to the drawbacks of centralized P2P systems, decentralized P2P systems emerged and are actually widely used. They rely on any central server and their all peers have equal status, rights and responsibilities. Each peer has only a partial view of the network and offers content (data/services) that might be relevant to only some queries peers. Thus, locating peers offering content quickly is a critical and challenging issue. The main advantages of such systems are: i) they do not have a single point of failure, ii) they can enjoy high scalability, performance, robustness, and other desirable features [Vu et al., 2010].

There are two logical network topologies (overlay) in the design of decentralized P2P systems: structured and unstructured. The difference between these two topologies lies in how queries are being forwarded among peers.

Unstructured P2P overlay is *"an overlay in which a node relies only on its adjacent nodes for delivery of messages to other nodes in the overlay. Example message propagation strategies are flooding and random walk"* [Buford et al., 2009]. In unstructured P2P overlays, each node is responsible for its own content, and keeps a registry of neighbors that it may forward queries to. Due to their simplicity, such overlays are pretty robust and withstand failures. However, they do not provide any mapping between the identifiers of contents and those of nodes [Vu et al., 2010]. This implies that: i) finding contents is challenging since it is difficult to predict which node maintains the queried content, ii) no guarantee on the completeness is provided, unless to search in the entire network, and iii) no guarantee on querying time is provided, except for the worst case (the entire network is searched). Among the most famous systems built using unstructured P2P overlays, we find Gnutella [Gnutella, 2012] and Freenet [Freenet, 2012].

Structured P2P overlay is: *"an overlay in which nodes cooperatively maintain routing information about how to reach all nodes in the overlay"* [Buford et al., 2009]. Compared to unstructured ones, structured overlays provide a limited number of query messages needed to find any content in the overlay. This is especially important when the content that we search is not popular or rarely available. To achieve this deterministic routing, nodes are placed into a virtual address space, the overlay topology is organized into a specific geometry (e.g., ring, toruse and hypercube), and a converging distance function that maps content and node identifier is defined for the routing algorithm [Buford and Yu, 2010]. To support these functionalities, most of the structured P2P systems rely on a Distributed Hash Table (DHT). A DHT is a particular instance of structured P2P overlays and is defined as follows: *"a DHT is a decentralized system that provides the functionality of a hash table, i.e., insertion and retrieval of key-value pairs. Each node in the system stores a part of the hash table. The nodes are interconnected in a structured overlay network, which enables efficient delivery of the key lookup and key insertion requests from the requestor to the node storing the key. To guarantee robustness to arrivals and departures of nodes, the overlay network topology is maintained and the key-value pairs are replicated to several nodes"* [Galuba and Girdzijauskas, 2009]. Every DHT defines its own key space. The P2P overlay uses the DHT keys for addressing its nodes. Each node has a specific location in the key space and stores the key-value pairs that are close to that location. The node's routing table is initialized when the node joins the overlay, using a specified bootstrap algorithm. Nodes periodically exchange their routing table (as part of overlay maintenance). Thus, a request can be routed to the node that maintains the desired content accurately and quickly. However, since the placement of content is tightly controlled, the cost of maintaining the structured topology of the overlay might be high, especially in a very large network environment [Vu et al., 2010].

3.3.2.3 Hybrid P2P systems

Centralized P2P systems are able to provide a reliable and quick content locating. However, the systems' scalability is affected due to the use of central server(s). Although decentralized P2P systems are more adapted to deal with this aspect than centralized P2P systems, they need

a longer time in content locating. Thus, to maintain scalability as in decentralized P2P systems, a hybrid design approach for P2P systems have been proposed.

The term hybrid is used in different disciplines and employed to characterize *"anything that is a mixture of two or more things"* [Cambridge Academic Content Dictionary, 2008]. In the context of P2P systems, this term is used to describe an approach that combines both centralized and decentralized architectures. Generally, hybrid P2P systems are realized using two kinds of peers (ordinary and super peers) and/or two hierarchical tiers (the upper tier serves the processing of the lower one) [Vu et al., 2010].

Super-peers are some peers that possess much more powerful capabilities and having more responsibilities than other (ordinary) peers. Super-peers form an upper-level in a hybrid system and are selected to act as servers for the ordinary peers such as in a centralized P2P system. Thus, ordinary peers can benefit from much more services, especially during the resource location process.

Hierarchical tiers in hybrid P2P systems are often in two levels. The upper level is dedicated to process some services for the lower level. For instance, the hybrid P2P system BestPeer [Ng et al., 2002] has a certain number of location independent global names lookup servers (LIGLOs) that serve as an upper level in the system. This upper level generates a unique identifier for the peers, helps peers to dynamically recognize their neighbors, and lets peers to reconfigure their neighbors with certain metrics. However, LIGLO does not provide resource location mechanisms.

In summary, hybrid P2P systems have several advantages and desirable functionalities that help peers to evolve easily into the network. For example, they can provide functionalities to optimize network topology, improve querying time and save system resource consumption.

Chapter 4

Discussion of related approaches

Contents

In this chapter, the main proposals of the scientific community related to the specification and analysis of software systems together with the execution in P2P architecture through middlewares are collected and discussed. Section 4.1 provides an overview of the most commonly used specification and analysis works, which are based on different approaches (like finite-state, process algebras and data-flow approaches) and proposed in various contexts (like CBSE, network protocol and database context). Section 4.2 does the same but for the most known works that propose runtime environments in P2P architecture.

4.1 Approaches for specification and analysis

The power of the software system analysis approaches depends on the modeling technique for the behavior of software systems. This behavior is usually modeled by Finite-State Automata (e.g., [Bures et al., 2008]). However, it may also be modeled by process algebras (e.g., [Allen and Garlan, 1997]), context-free languages (e.g., [Burkart and Steffen, 1992]), pushdown processes (e.g., [Burkart and Steffen, 1994]), etc.

In the context of the component-based system, the finite state approaches usually use regular languages to describe component behavior. However, these finite state approaches can only

handle bounded recursion (i.e., up to a certain depth) and limited abstraction of the data-flow. To address this more explicitly, we discuss hereafter some of such approaches.

There is a large body of component models using various formal and semi-formal specifications in the context of component-based systems. These specifications concentrate on different aspects of component modeling. Due to this diversity, we refer to [Rausch et al., 2008] which provides an interesting study of state-of-the-art in component-based systems. Among the component models discussed in [Rausch et al., 2008], KobrA [Atkinson et al., 2008] is a UML-based method for describing components and component-based systems. It uses different diagrams representing three projections: structural, functional and behavioral. KobrA is not a formal language, but rather a set of principles for using mainstream modeling language. It provides a certain degree of flexibility because anything that conforms to its principles can in practice be accommodated within the method. Rich Services [Demchak et al., 2008] provides an architectural framework that reconciles the notion of service with hierarchy (systems-of-systems). It uses message sequence charts in order to specify component behaviors. This allows the approach to model bounded recursion. rCOS [Chen et al., 2008] is an extended theory of UTP (Unifying Theories of Programming) [Hoare and Jifeng, 1998] for object-oriented and component-based programming. UTP combines the reasoning power of predicate calculus with the structuring power of relational algebra. In rCOS approach, each component interface has a contract. A contract only specifies the functional behavior in terms of predicates (pre and post conditions) and a protocol defining the acceptable traces of method calls. The behavior is specified by a state diagram and should be accepted by FSA. The protocol is specified by a sequence diagram. The reasons for having these two diagrams are different. In fact, the sequence diagram allows generating CSP processes to deal with concurrency, when the state diagram has an operational semantics which is easier to use for verification with model checking. SOFA [Bures et al., 2008] is a hierarchical component model. It is dedicated to the development of distributed application with dynamic update of components. It uses *behavior protocols* for the specification of interaction behavior of components. This allows to verify the system architecture independently from the implementation, and the relation of the component model and implementation. In order to fully automate behavior verification, a tool chain is used. It consists of *behavior protocols* to Promela translator and the Spin model checker. However, *behavior*

protocols cannot treat behavior that cannot be specified by a regular language. Like SOFA, Factal also uses *behavior protocols* to specify component behavior. Therefore, they have the same limitation on the description of component behavior.

Since the finite state models are not providing an adequate abstraction of a system that contains recursive call-backs, context-free model checking have been proposed. Among the first works in this direction, we could mention [Burkart and Steffen, 1992], which presents an algorithm that decides whether a property written in the alternation-free modal mu-calculus is satisfied for context-free processes, i.e., for processes that are given in terms of a context-free grammar or equivalently. In [Burkart and Steffen, 1994], pushdown processes are proposed as a generalization of context-free processes to better support parallel composition. Pushdown processes are processes that can be (finitely) represented by means of classical Pushdown Automata. After introducing these two approaches, several models [Alur et al., 2005, Benedikt et al., 2001, Esparza et al., 2000, Burkart and Steffen, 1997] for infinite-state systems have been proposed especially to decrease checking complexity. But in the end, these models are still closely related to context-free processes and pushdown processes, and usually have the same expressiveness. In contrast to our approach, they cannot handle recursive call-backs which gives rise to context-sensitive structures (cf. Section 2.2.1).

Process algebras such as CSP (Communicating Sequential Processes) or CCS (Calculus of communicating systems) can be used as an alternative approach for verifying protocol conformance. These algebraic approaches are more powerful than FSA and context-free grammars. According to Milner [Milner, 1980], algebra appears to be a natural tool for expressing how systems are built. However, in order to automate analysis, some constraints on the specification language can be required. For instance, in [Allen and Garlan, 1997], the authors have been restricted their use of the CSP notation in a way that processes will always be finite. Therefore, the analysis of the behavior boiled down to a finite-state verification.

Compared to other works where component approach is dedicated to manipulate protocols, Reussner [Reussner, 2002] presents the model of counter-constrained finite state machines. It is an extension of finite state machines, specifically created to model protocols containing dependencies between services due to their access to shared resources. However, Reussner does

not consider composition operators and does not provide an underlying discipline. Puntigam [Puntigam, 2003] shows that it is possible to develop component interfaces specifying non-regular protocols for the communication between components and the rest of a system. The concepts proposed in this paper need support from a programming language. However, no practically usable programming language supports these concepts.

Different data-flow based approaches have been proposed in the domain of system modeling. In [Garousi et al., 2005], a control-flow analysis for UML 2.0 sequence diagrams is presented. To define the control-flow, the authors propose an extended activity diagram meta-model. [Yang et al., 2009] presents DFA-based algorithms to analyze BPEL programs and detect their data-flow anomalies. These algorithms operate on a control-flow graph derived from Activity Object Tree (AOT). The AOT is based on Eclipse Modeling Framework to express the relationships among activities. [Zhou and Lee, 2006] proposes a causality interface for deadlock analysis in a concurrent model of computation called *Dataflow*. It shows that deadlock is decidable for synchronous *Dataflow* models with a finite number of actors. [Cain et al., 2008] presents an approach where a meta-model of an object oriented program's runtime is constructed to manage DFA. This meta-model contains classes that represent the relationship between the program elements (e.g., classes, objects and methods) in order to create an abstract representation for DFA. Like these different approaches, we also use DFA-based algorithms to analyze the constructed systems. However, our approach is dedicated to component-based P2P applications. It provides a formalism to capture their specific behavior (i.e., recursive callbacks) and constructs an abstract representation (i.e., DDG) from which we can obtain multiple implementations of the control logic by analyzing the order of data evaluation.

The principle of the transformation of an abstract representation is also present in other formal systems. Many of those formal systems, such as λ-calculus [Sheard, 1997], catamorphisms [Launchbury and Sheard, 1995], hylomorphisms [Onoue et al., 1997] and other from category theory, have been studied in previous works of Parigot (e.g., [Correnson et al., 1999]) and a large comparison of these different formal systems can be found in [Duris, 1998] [Correnson, 2000]. These works show that those formal systems share a similar global structure. They abstract programs in some mathematical domain. Then, the transformed program is obtained by a backward translation from the mathematical domain. For instance, the HYLO system [Onoue et al., 1997]

transforms a program into hylomorphisms and then performs partial data evaluation. After that, these new hylomorphisms could be translated back into a program. However, these formal systems share a surprising constraint: the abstraction always relies on objects where recursive structures or schemes are strongly preserved and cannot be easily modified. For example, with λ-calculus, the recursive calls are altogether defined in the structure of the λ-terms. With hylomorphisms, these recursion schemes are exactly pointed out by functors (a special type of mapping in category theory) which are used as transformation parameters. Thus, transformations cannot freely restructure the abstract representation. Taking in mind these previous studies, DDF has then been defined with the following distinctive characteristics: i) allowing parts of the control logic (even if it is recursive) to be described conceptually separated from other parts by using the concept of rules; ii) the user describes what is to be done rather than the details of how it is to be done; iii) from a single specification, multiple implementations can be synthesized by analyzing the order of data evaluation.

Other works relevant in the context of our approach can be found in database and network protocol communities. They are applied, for example, in [Alvaro et al., 2010] to the Cloud Computing in order to raise the level of abstraction for programmers and improve program correctness in a data-centric, declarative style. Another interesting approach is P2 [Loo et al., 2005]. It can be viewed as a synthesis of ideas from these two communities works applied to overlay networks [Andersen et al., 2001]. P2 is a system that uses a declarative logic language to express and implement overlay networks. It directly parses and executes such specifications into a data-flow program. The approach proposed in [Lin et al., 2011] seems to be close to our work in the sense that it also passes through the construction of a dependency graph to perform some optimizations. The difference between those works and our approach is that they are not based on components, what usually drives them to specify into their models (e.g., relational algebras and rule-based specification) the whole application.

4.2 Execution in P2P architecture through middlewares

There has been a large body of related work carried out to develop P2P middlewares. This has proposed increasingly novel approaches addressing application from many different domains

such as distributed sharing of data, video streaming and gossip communications. For example, JavaPorts framework [Manolakos et al., 2001] aims to provide a set of tools that will enable developing parallel applications on a network of heterogeneous workstations. A JavaPorts application can be defined as a collection of interacting tasks using a Task Graph abstraction. In this graph the nodes correspond to application Tasks. Tasks communicate using point-to-point connections between peer ports. Expeerience [Bisignano et al., 2003] is a middleware providing support for mobile application developers exploiting P2P technology over ad hoc networks. It has been developed in Java and is based on JXTA. It manages the discovery service, multiple interfaces, intermittent connectivity and code mobility. SpiderNet [Gu et al., 2004] is a P2P service composition framework. It achieves service composition by supporting directed acyclic graph composition topologies and considering exchangeable composition orders. SpiderNet provides failure recovery scheme that maintains a small number of dynamically selected backup compositions to achieve quick failure recovery for realtime streaming applications. Juno [Tyson et al., 2008] is a networking middleware dedicated to multimedia content distribution (e.g., file sharing, video on demand and live streaming). It is designed in a component-based manner and has been implemented using the OpenCOM [Coulson et al., 2008] component model. Juno provides a configurable framework, allowing the middleware to be specialised and adapted to a variety of environments. Kompics [Arad and Haridi, 2010] is a message-passing component model that can be used for building P2P systems. Kompics provides a framework to compose protocol layers in a similar way to Mace [Killian et al., 2007] and Wids [Lin et al., 2005]. Mace is a language support for building distributed systems as C++ components. It allows describing each layer of the distributed system as a reactive state transition model. This state transition model enables model checking of the system implementation to find both safety and liveness bugs. WiDS is a toolkit that provides several run-times to run P2P protocols in different modes. In particular, in its simulation engine that helps to evaluate and debug P2P protocols in a controllable environment.

The main characteristics that distinguish SON from the approaches outlined above can be summarized as follow:

- Son's applications are specified by a rule-based language that captures the recursive behavior of P2P applications. This kind of behavior is very frequent in the context of P2P

and many modeling approaches cannot describe it.

- More general abstraction for P2P applications can be induced from the specification rules. This abstraction represents only one data-flow model (dictated by data dependencies) on the execution. Further, it exposes the right level of detail to perform DFA.
- SON ensures that the target implementation and generated code fit well the behavioral constraints contained in the specification rules.
- SON's user implements only the code corresponding to the declared services. Afterwards, a code generation tool generates the containers of the components. The component container embodies all resources needed to adapt the implementation code to the run-time environment.
- SON can be considered as a generic lightweight middleware (with the necessary set of operations that must be present to develop any kind of component-based P2P applications) for the following reason. Since, in most cases, the challenges of P2P systems can be reduced to a single problem: *"How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy?"* [Balakrishnan et al., 2003], SON has been unified the notion of publish/subscribe: it uses a DHT not only to publish and subscribe data, but also to enable dynamic service publication, discovery, and deployment.

Part III:

Our proposal

Chapter 5

DDF: A formal language to specify component-based P2P applications

Contents

This chapter introduces a formal language to specify and analyze component-based P2P applications. It is called DDF (Data Dependency Formalism) and used as an underlying formalism for the work presented in this thesis. DDF has been essentially developed for the following two reasons. Firstly, to formally describe the recursive behavior of P2P applications. This kind of behavior is very frequent in the context of P2P and many modeling approaches cannot describe it, as explained in Section 2.2.1. Secondly, to formally construct an abstract representation (i.e., Data-Dependency Graph) for P2P applications. This abstraction exposes the right level of detail to perform data-flow analyzes. Throughout this chapter, the DDF concepts are illustrated and explained on a number of examples distilled from a case study that consists of a gossip

protocol [Voulgaris et al., 2005, Jelasity et al., 2007]. Gossip protocols, also called epidemic protocols, are well-known in the community of P2P to ensure a reliable information dissemination. This kind of dissemination is a common behavior of various P2P applications, and according to [Jelasity, 2011], a large number of distributed protocols can be reduced to a gossip protocol. Before presenting this case study and then the DDF specification, a short explication is provided to show why Attribute Grammars have inspired us to develop DDF.

5.1 Why our formalism is inspired by the Attribute Grammars

Many techniques and algorithms for Data-Flow Analysis (DFA) were introduced in Attribute Grammars (AGs) literature. These techniques and algorithms are commonly used in compiler construction for performing optimizations from a program's abstract representation (an attribute-dependency graph induced by the Abstract Syntax Tree of the program). In a previous work of [Parigot et al., 1996], it has been argued that in the term "Attributed Grammar" the notion of *grammar* does not necessarily imply the existence of an underlying tree, and that the notion of *attribute* does not necessarily mean decoration of a tree. Hence, Dynamic Attributed Grammars (DAGs) have been presented by Parigot et al. as an extension to the AG formalism. DAGs are consistent with the general ideas underlying AGs, thing that allows them to retain the benefits of the results that are already available in that domain. In the same direction, we define our formalism, which will allow us to construct a Data-Dependency Graph (DDG) for component-based P2P applications and use the already developed DFA algorithms to perform analyzes on it. To achieve this, we have inspired by the main characteristics of AGs and DAGs. Those characteristics are briefly presented in the rest of this section.

Structural decomposition and declarative character

AGs were introduced by Knuth [Knuth, 1968] and, since then, they have been widely studied [Deransart et al., 1988, Deransart and Jourdan, 1990, Paakki, 1995]. An AG is a declarative specification that describes how attributes (variables) are computed for rules in a particular syntax (i.e., it is syntax-directed). They were originally introduced as a formalism for describ-

ing compilation applications; they were intended to describe how to decorate a tree, and could not be easily thought about in the absence of the structure (tree) representing the program to compile. In this application area, AGs were recognized as having these two important qualities:

- they have a natural *structural decomposition* that corresponds to the syntactic structure of the language, and
- they are *declarative* in that the writer only specifies the rules used to compute attribute values, but not the order in which they will be applied.

Thus, a program can be described by AGs as a set of productions. Each production p describes an elementary control-flow and has the following form:

$$p : X_0 \rightarrow X_1, ..., X_n$$

X_0 represents a node in a tree and $X_1, ..., X_n$ are its child nodes. For each production p we give a set of *semantic rules* defining the computation of the *synthesized* attributes of X_0 and the *inherited* attributes of $X_{1 \leq i \leq n}$. The *synthesized* attributes are the result of the attribute computation, and may use the values of *inherited* attributes. *Synthesized* attributes are used to pass computed information up the tree, while *inherited* attributes pass information down and across it.

Synthesized and inherited attributes

To illustrate the concept of *synthesized* and *inherited* attributes, Table 5.1 gives AG productions that describe the computation of terms like $3 * 5$ and $3 * 5 * 7$. In this example (extracted from [Aho et al., 1986]), each of the nonterminals T and F has a synthesized attribute *val*; the terminal **digit** has a synthesized attribute *lexval*, which is an integer value returned by the lexical analyzer. The nonterminal T' has two attributes: an inherited attribute *inh* and a synthesized attribute *syn*. The *semantic rules* are based on the idea that the left operand of the operator $*$ is inherited. More precisely, the head T' of the production $T' \rightarrow * F T'_1$ inherits the left operand of $*$ in the production body. Given a term $x * y * z$, the root of the subtree for $* y * z$ inherits x. Then, the root of the subtree for $* x$ inherits the value of $x * y$, and so on, if there are more factors in the term. Once all the factors have been accumulated, the result is passed back up the tree using synthesized attributes.

To see how the semantic rules are used, consider the annotated parse tree for $3 * 5$ in Figure 5.1. The leftmost leaf in the parse tree, labeled digit, has attribute value $lexval = 3$, where the 3 is supplied by the lexical analyzer. Its parent is for production 4, $F \rightarrow$ **digit**. The only semantic rule associated with this production defines $F.val =$ **digit**.$lexval$, which equals 3.

At the second child of the root, the inherited attribute $T'.inh$ is defined by the semantic rule $T'.inh = F.val$ associated with production 1. Thus, the left operand, 3, for the $*$ operator is passed from left to right across the children of the root.

The production at the node for T' is $T' \rightarrow * F\ T'_1$ (we retain the subscript 1 in the annotated parse tree to distinguish between the two nodes for T'). The inherited attribute $T'_1.inh$ is defined by the semantic rule $T'_1.inh = T'.inh \times F.val$ associated with production 2.

With $T'.inh = 3$ and $F.val = 5$, we get $T'_1.inh = 15$. At the lower node for T'_1, the production is $T' \rightarrow \epsilon$. The semantic rule $T'.syn = T'.inh$ defines $T'_1.inh = 15$. The syn attributes at the nodes for T' pass the value 15 up the tree to the node for T, where $T.val = 15$.

Productions		Semantic rules
(1)	$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
(2)	$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
(3)	$T' \rightarrow \epsilon$	$T'_1.syn = T'.inh$
(4)	$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$

Table 5.1: Attribute Grammar productions of a simple multiplication calculator.

Dependency graph

A dependency graph is used to determine an evaluation order for the attribute instances in a parse tree. In other words, it helps to determine how the values of attributes can be computed. An important number of DFA algorithms introduced in AGs literature are based on it. An edge in a dependency graph from one attribute instance to another indicates that the value of the first

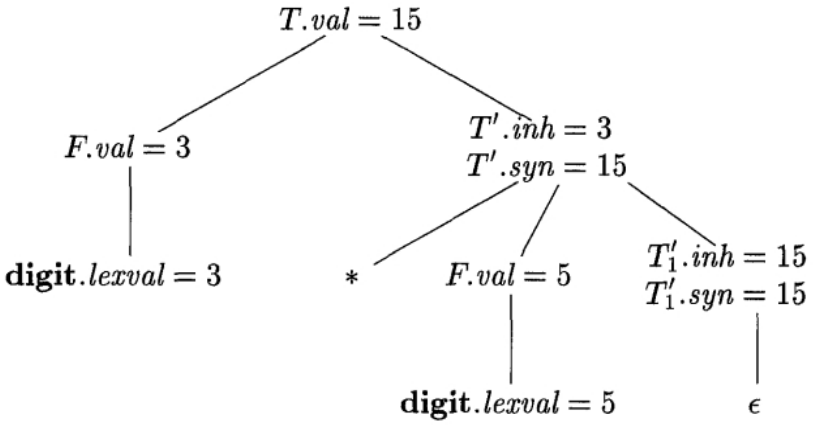

Figure 5.1: Annotated parse tree for $3 * 5$. *(from [Aho et al., 1986])*

is needed to compute the second. This allows to express the constraints implied by the semantic rules. To illustrate that, we consider the same example extracted from [Aho et al., 1986] and we present in Figure 5.2 the dependency graph for the annotated parse tree of Figure 5.1. The nodes of this dependency graph, represented by the numbers 1 through 9, correspond to the attributes.

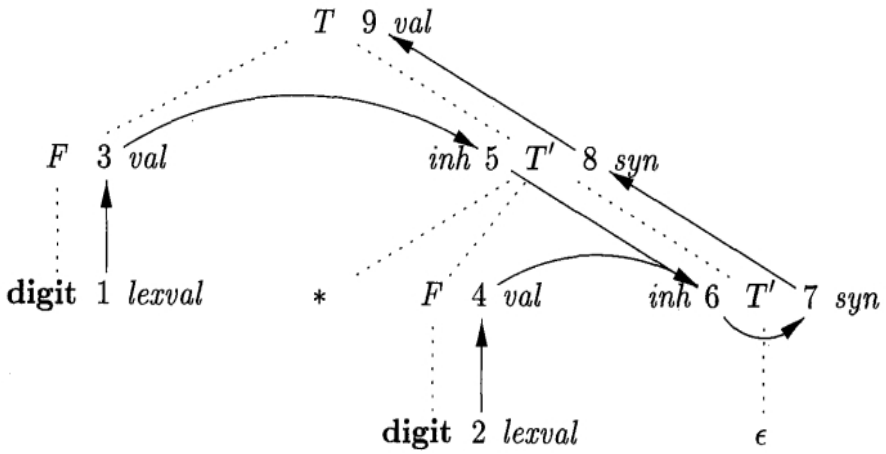

Figure 5.2: Dependency graph for the tree of Figure 5.1. *(from [Aho et al., 1986])*

Nodes 1 and 2 represent the attribute *lexval* associated with the two leaves labeled **digit**. Nodes 3 and 4 represent the attribute *val* associated with the two nodes labeled F. The edges to node 3 from 1 and to node 4 from 2 result from the semantic rule that defines $F.val$ in terms of **digit**.*lexval*. In fact, $F.val$ equals **digit**.*lexval*, but the edge represents dependence, not equality. Nodes 5 and 6 represent the inherited attribute $T'.inh$ associated with each of the occurrences of nonterminal T'. The edge to 5 from 3 is due to the rule $T'.inh = F.val$, which defines $T'.inh$ at

the right child of the root from $F.val1$ at the left child. We see edges to 6 from node 5 for $T'.inh$ and from node 4 for $F.val$, because these values are multiplied to evaluate the attribute inh at node 6. Nodes 7 and 8 represent the synthesized attribute syn associated with the occurrences of T'. The edge to node 7 from 6 is due to the semantic rule $T'.syn = T'.inh$ associated with production 3 in Table 5.1. The edge to node 8 from 7 is due to a semantic rule associated with production 2. Finally, node 9 represents the attribute $T.val$. The edge to 9 from 8 is due to the semantic rule, $T.val = T'.syn$, associated with production 1.

Description of recursion and conditions with Dynamic AGs

Because of their historical roots in compiler construction, the notion of (physical) tree in AGs was considered as the only way to direct computations. This is the main cause for their lack of use and lack of expressiveness. Some works have attempted to respond to this problem by proposing extensions to the classical AG formalism, for instance Higher-Order Attribute Grammars [Swierstra and Vogt, 1991], Circular Attribute Grammars [Farrow, 1986], Multi-Attribute Grammars [Attali, 1988] or Conditional Attribute Grammars [Boyland, 1996]. The main difference between these works and the one proposed by [Parigot et al., 1996] is the methodology used to attack the problem. All of them, in a first step, propose a linguistic extension designed to make the expression of a particular application easier (for instance, data-flow analysis for Circular AGs) and, in a second step, wondered how this extension could be implemented. In contrast, the approach [Parigot et al., 1996] was, first, to precisely characterize the intrinsic power of the classical formalism and, thereafter, to derive the language extensions that allow to fully exploit this power. The basic observation is that the notion of *grammar* does *not* necessarily imply the existence of a (physical) *tree*. In fact, the proposed view of the grammar underlying an AG is similar to the grammar describing all the call trees for a given functional program or all the proof trees for a given logic program: the grammar precisely describes the various possible flows of control. In this context, a production describes an elementary recursion scheme (control flow), whereas the semantic rules describe the computations associated with this scheme (data flow).

It is very important at this point to observe that all the theoretical and practical results on AGs (in particular, the algorithms for constructing efficient attribute evaluators) are based *only*

on the abstraction of the control flow by means of a grammar and not at all on how its instances are obtained at run-time. In consequence, two notions which comply with this view have been presented by Parigot et al.:

- *Grammar Couples* allow to describe recursion schemes independently from any physical structure and/or to exhibit a different combination of the elements of a physical structure. A grammar couple defines an association between a dynamic grammar and a physical or concrete grammar.
- *Dynamic Attribute Grammars* (DAGs) are defined on top of *Grammar Couples*. They allow attribute values to influence the flow of control by selecting alternative dynamic productions.

These extensions eliminate the major criticism against AGs, namely, their lack of expressiveness. As an example to illustrate that, let's see how to describe the structure and dynamic semantics of the **while** statement. If `STAT,COND` respectively represent statements and boolean conditions, Table 5.2 shows the productions for the **while** statement. In this example, `name:TYPE` means that `TYPE` is the type of `name`. $p \in P_c$ is the concrete production which describes that a **while** statement is made of a condition and a body statement. p_r and $p_t \in P_d$ are two dynamic productions which respectively represent the recursive behavior of a **while** structure (p_r) as long as the condition is true and the termination case (p_t) when the condition becomes false.

Concrete production $p \in P_c$:

p : `while : STAT` $\rightarrow$ `cond : COND  body : STAT`

Dynamic productions p_r and $p_t \in P_d$:

p_r : `w = while : STAT` $\rightarrow$ `cond = cond : COND`
`body = body : STAT  w-rec = while : STAT,`

p_t : `w = while : STAT` $\rightarrow$ `cond = cond : COND`

Table 5.2: Part of a grammar couple for the **while** statement

Table 5.3 presents the *semantic rules block* describing the conditional semantic of our example of the **while** statement. Attributes names are prefixed by $h.$ for inherited, and $s.$ for

synthesized. The attribute *env* represents the execution environment of a statement and *s.c* carries the value of the boolean condition.

```
{ h.env(cond) := h.env(w),                                  // common semantic rule
  { (s.c(cond)),                                            // boolean expression
    { (w = while : STAT → cond = cond : COND
                                          body = body : STAT w-rec = while : STAT,
      h.env(body) := h.env(w)                               // true case:
      h.env(w – rec) := s.env(body)
      s.env(w) := s.env(w-rec) },
    { w = while : STAT → cond = cond : COND,                // false case:
      s.env(w) := h.env(w) } } }
```

Table 5.3: The semantic rules block for the **while** statement

Attribute Grammars and our Data Dependency Formalism

Although AGs were introduced long ago, their lack of expressiveness has resulted in limited use outside the domain of static language processing. With the notion of *Dynamic Attribute Grammars* defined on top of *Grammar Couples*, it is possible to extend this expressiveness and to describe computations on structures that are not just trees, but also on abstractions allowing for infinite structures. In our work, we explore to take advantage of this to define a Data Dependency Formalism. DDF is consistent with the general ideas underlying AGs. Hence, we expect to retain the benefits of the results and techniques that are already available in AGs' domain, in particular, those introduced for Data-Flow Analysis.

5.2 Case study: Gossip protocol

As presented in Section 2.2.4, a large number of algorithms and protocols have been developed for P2P applications to support different properties. These algorithms and protocols usually have a recursive behavior that many modeling approaches cannot describe and analyze. To illustrate how our approach can deal with this issue, we have chosen to explain our Data Dependency Formalism in a case study that consists of a gossip protocol [Voulgaris et al., 2005]

[Jelasity et al., 2007]. In addition to the fact that gossip protocols present a recursive behavior, our choice is especially motivated by the following other reasons. According to [Jelasity, 2011], a broad range of distributed protocols can be reduced to gossip protocols, and those gossip protocols can help in the building of large-scale cloud computing systems, which are considered the computing platform of the future by many actors in both research and industrial communities. The rest of this section presents such kind of protocols in more detail.

Gossip in human communities

Humans frequently try to find information about those around them. But interconnections in societies are complex, and it is impossible to be present at the same time in different places to get this kind of information directly. Therefore, people pick it up through an intermediary, whether or not they have the possibility and patience to confirm it later either directly or indirectly. This phenomenon, called gossip (or rumor, which differs primarily by being speculative and sometimes pertaining to events rather than people), is an important social behavior that nearly everyone experiences, contributes to, and presumably intuitively understands [Foster, 2004].

To complement views of gossip as essentially a means for spreading and gaining information, [Baumeister et al., 2004] proposes that gossip helps people learn about how to function effectively within the complex and ambiguous structures of human social life. Gossip can thus be understood as an extension of observational learning, in the sense that people can learn from the success and failures of others outside of one's field of vision and sometimes even outside one's circle of friends.

Gossip and epidemics

The first real application of gossip as a protocol in the context of computer systems was presented in [Demers et al., 1987]. In this paper, the authors recognize its power of spreading information and propose a formal treatment to ensure that each replica of the database on the Xerox Corporate Internet[1] (CIN) was up-to-date. They were originally inspired by the way in which epidemics spread in a biological community. Thing that is closely analogue

[1]The worldwide CIN comprised several hundred routers connected by gateways and phone lines of many different capacities. It also comprised thousands of workstations, servers and computing hosts.

to gossip. In fact, disease epidemics are contagious and spread from person to person when a virus (that plays the role of a piece information) enters the body. Hence, the term *epidemic algorithm/protocol* is sometimes employed when describing a computer system in which gossip-based information dissemination is used. Figure 5.3 based on the one proposed by [Eugster et al., 2004] illustrates that. In this figure, a multicast source, represented by the black circle, sends a message to be disseminated in a system of size n. Each infected process—each process that receives the message—forwards it by default to a randomly chosen subset of other processes. Afterward, each of these infected processes in turn forwards the information to another random subset. Eventually, the message will reach all processes of the system with a high probability after a certain number of rounds.

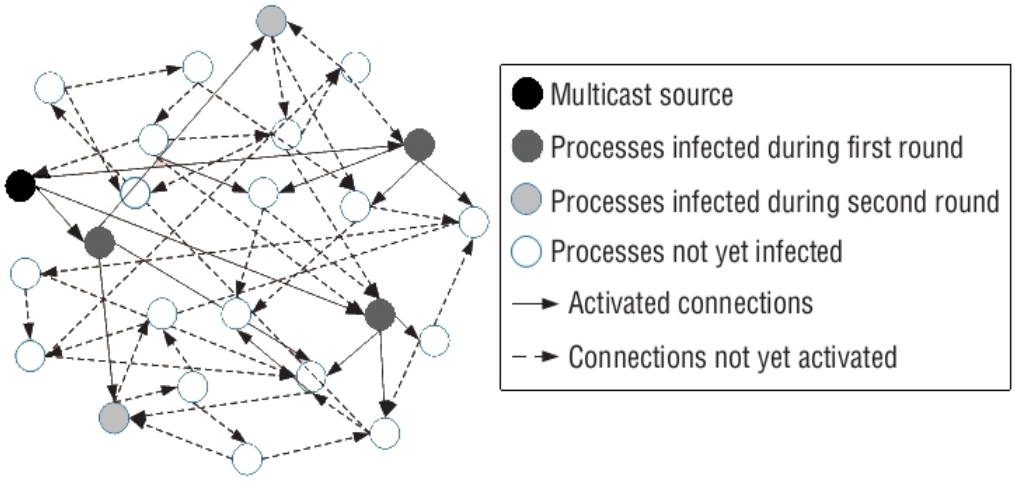

Figure 5.3: Epidemic algorithm *(based on a figure from Eugster et al. 2004)*.

Applications of gossip in computer systems

In the last years, epidemic/gossip protocols have been widely used to exchange information (data) in large-scale P2P systems [Jelasity et al., 2007]. This has been motivated by the capacity of these protocols to ensure that information is reliably exchanged, even if the peers dynamically join and leave the system, or the underlying network suffers from broken or slow links.

Beyond disseminating information in distributed systems, gossiping can be generalized to different applications for various domains such as resource management, overlay maintenance

and data aggregation. For instance, in the context of data aggregation, in which information about a large distributed system needs to be gathered and expressed in a summary form, gossiping can be used as an efficient tool for computing aggregates, e.g., sums, averages and maximum of some attribute of the system nodes. We refer to [Kermarrec and van Steen, 2007] for a survey on gossiping applications.

Gossip algorithm

In [Jelasity et al., 2007], the authors present a generic and simple gossip algorithm, which factors out the very abstraction of a peer-sampling service and captures many possible variants of gossip-based protocols (the simple template presented in the Introduction relies on this algorithm). For these reasons, we adopt exactly the same formulation of this gossip algorithm, which we express using DDF in the next section.

In a gossip algorithm, each node in the network periodically exchanges information with a subset of other nodes. In fact, every node maintains a local membership table providing a partial view on the complete set of memberships and periodically refreshes the table using a gossiping procedure. The table (view) is a list of c node descriptors, where c is the size of the list and is the same for all nodes. A node descriptor contains a node network *address* and an *age* that represents the freshness of the given node descriptor. The list changes by means of usual list operations (e.g., permute) that are defined on it. Therefore, the tables reflect the dynamics of the system by continuously changing random subset of the nodes (in the presence of failure and joining and leaving nodes).

The algorithm consists of two activities (serving and consuming) in each node: an active client gets activated in each T time units exactly once and then initiates communications with other nodes, and a passive server waits for and answers these requests. The behavior of the passive server just mirrors the same stapes of the active client. In terms of DDF, each activity corresponds to a pair of rules given in table 5.7. This table describes the behavior of a *Gossip System* constituted of two nodes ($node_x$ and $node_y$) and the associated methods (implementations) extracted from [Jelasity et al., 2007]. The detailed description of this system is formally defined with DDF and presented in progressive manner throughout the next section.

5.3 DDF specifications

Our Data Dependency Formalism (DDF) is essentially dedicated to applications that can be divided into autonomous components communicating to each other over channels. For this purpose, we separate clearly computational activities and component interactions. Thus, we distinguish two types of descriptions, grouped as syntactic and semantic descriptions. The syntactic descriptions consist of a collection of input, output and internal services described only by their signatures. The semantic descriptions consist of interaction rules that define not only the valid sequences of service invocations, but also data exchange required for achieving of the functional activities and driven the interactions between components. We call *interface* the syntactic part and *behavior* the semantic part.

5.3.1 Interface

A service is a functional activity supported by a component. If the component provides a service through its interface, the service is called input service; if the component requires a service through its interface, the service is called output service. If the component provides a service that is invoked only by itself, the service is called internal service. In a component, a service call refers to an output service or an internal service.

An internal service represents a particular action of a component. To describe, for example, time sequence (one component's behavior occurs after some time), an internal service $timer(timeout : Int)$ can be used to represent a timer. This internal service *timer* has an argument *timeout* that can be set as an integer. Once $timer.timeout$ is set, the component's behavior can only occur when $timer.timeout == 0$.

Formally, a service and an interface are defined as follow:

Definition 1 (Service). *A service is a 3-tuple* $\delta =< type, name, arg >$*, where:*

- *type is the service type;*
- *name is the service name;*
- *arg is a set of the service arguments.* □

A service s is written as $s(a_0, ..., a_n)$, its result is denoted by $s\$$ and its arguments are denoted by arg_s with $arg_s = (a_0, ..., a_n)$.

Definition 2 (Interface)**.** *An interface is a 3-tuple* $I =< S_{in}, S_{out}, S_{int} >$*, where:*

- S_{in}, S_{out}, S_{int} *are a set of, respectively, input, output and internal services.* □

Example 1 (Interface of a gossip component)**.** According to Definition 2, the interface of a gossip component (called *Node*) is expressed as $I_{Node} =< S_{in}, S_{out}, S_{int} >$, where:

- $S_{in} = \{gossip(buffer : Buffer), answer(buffer : Buffer)\}$;
- $S_{out} = \{answer(buffer : Buffer), gossip(buffer : Buffer)\}$;
- $S_{int} = \{timeout(T : TimeUnit)\}$. □

A gossip component has two activities: serving and consuming information (data). The two input services are for the serving activity and the two output services are for the consuming activity. The behavior of input services (serving activity) just mirrors the same stapes of output services (consuming activity). The *gossip* service is for the propagation of data and the *answer* service is for sending a response to the sender. Figure 5.4 illustrates these features.

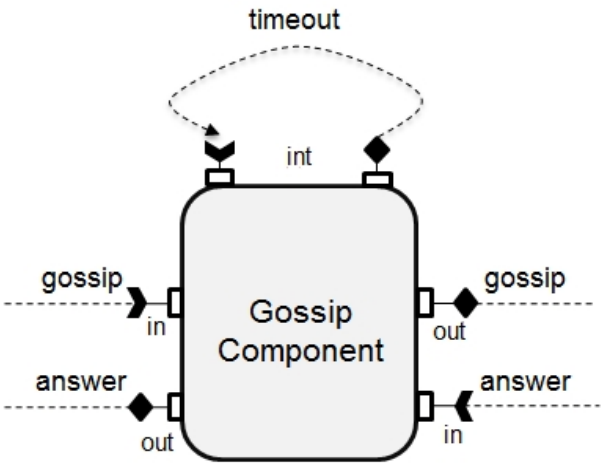

Figure 5.4: Services of a gossip component.

5.3.2 Component

A component encapsulates data (attributes) with methods to operate on the component's data. Methods implement the services provided through the component interface. A service is implemented by one method. A component contains the declaration of attributes whose values define

the state of its instances, along with the bodies of methods that operate on those attributes. A method defined within a component can access only those attributes that are declared within the component, along with any arguments that are passed to the method.

The component prohibits concurrent access to its methods. Only one method can be run within the component at any one time. Consequently, the programmer does not need to code this synchronization explicitly; it is built into the component. This technique is widely used in operating systems [Silberschatz et al., 2008] to simplify reasoning about the implementation of concurrent distributed applications.

During run time, a component might need inputs. When it receives an input, the component will respond to this by executing its methods and/or changing its state (attributes). Otherwise, without inputs, a component may produce an output and/or change its states. This output may have an eventual response as an input.

Formally, a component is defined as follows:

Definition 3 (Component). *A component is a 4-tuple* $C =< A, I, Imp, m >$*, where:*

- A *is a set of typed attributes;*
- I *is an interface;*
- Imp *is a set of methods (implementing the services provided through the interface). A method is denoted* F *and defined in Definition 6;*
- $m : \{S_{in}, S_{out}\} \rightarrow Imp$ *is a function that maps each service* $s \in (S_{in} \cup S_{int})$ *of* I *to a component method in* Imp*.* □

An attribute may be chosen as a component state. State changes are caused by an input, output or internal service. Thus, for the external environment, the input and output services may describe a visible state change. These states may be used by guard conditions (defined in Section 5.3.3) to control the component behavior.

A component may have multiple instances. An instance c_i of a component $C = (A_C, I_C, Imp_C, m_C)$ is denoted by $c_i : C$.

Example 2 (Gossip component)**.** According to definition 3, a gossip component, called *Node*, is expressed as $Node =< A, I, Imp, m >$, where:

- $A = \{view : List(IP : Address, age : Int),\ buffer : List,\ c : Int,\ push : Bool,\ pull : Bool,$
  $T : Time,\ H : Int,\ S : Int\}$;
- $I = I_{Node}$;
- $Imp = \{F_{r_1}(), F_{r_2}(), F_{r_3}(), F_{r_4}()\}$;
- $m : \{S_{in}, S_{out}\} \rightarrow Imp$. □

An instance of the gossip component *Node* has an *IP* address to exchange services with other instances of *Node* in a P2P network. Each instance maintains a *view* representing its partial knowledge of the network membership. A *view* is a list of c couples (IP, age). Attribute c represents the size of the *view* and is the same for all instances. A couple (IP, age) contains an *IP* address of an instance in the network and an *age* that represents the freshness of this instance.

To reflect the dynamics of the system (joining and leaving instances), the gossip algorithm (executed periodically on each instance and implemented by the methods $\{F_{r_1}(), F_{r_2}(), F_{r_3}(), F_{r_4}()\}$ that we explain later in this Chapter) updates the views by changing their random subset of the instances.

In fact, once a running instance selects another instance to exchange membership information with and the information has to be pushed (boolean attribute *push* is true), then the *buffer* of the running instance is initialized with a fresh information ($IP = Myaddress, age = 0$). Then, $c/2 - 1$ elements are selected randomly from the *view* (ignoring the oldest ones) and appended to the *buffer*. The number of the oldest elements (as defined by their age) is H and is less than or equal to $c/2$. H defines how aggressive the gossip algorithm should be when it comes to removing links that potentially point to faulty instances (dead links). In other words, if an instance is not alive, then its information will never get refreshed (and thus become old), and therefore sooner or later it will get removed. The larger H, the sooner older elements will be removed from the *view*. The *buffer* created this way is sent to the selected instance. If a reply is expected then the boolean attribute *pull* is true. After removing the H oldest elements, the S first elements are removed from the *view*. These S elements are exactly the ones that were

sent to the instance previously. If S is high, then the received elements will have a higher probability to be included in the new view. Since the same gossip algorithm is run on the receiver side, this mechanism in fact controls the number of elements that are swapped between the two instances. If S is low then both parties will keep many of their exchanged elements. We present in Table 5.4 a summary of these attributes' features and we refer to [Jelasity et al., 2007] for more details.

Attributes	Explanations
$view(IP, age)$	is a list of couples. Each couple contains an IP address of a node in the network and an age that represents the freshness of this node.
$buffer$	is a temporal list used to store output or input information.
c	is the size of the $view$.
$push$	if it is true, then the information will be sent to the selected instance.
$pull$	if it is true, then a reply is expected.
H	is the number of the oldest elements in the $view$ and is less than or equal to $c/2$. H defines how aggressive the algorithm should be when it comes to removing links that potentially point to faulty instances.
S	is the number of elements that were sent to the selected instance. If S is high, then the received elements will have a higher probability to be included in the new view.

Table 5.4: Component attributes.

5.3.3 Behavior with data dependency

As in the grammar-based modeling methods which are well suited to describing the control logic for the processing of data streams [Börger, 2000], the aim of our specification is to describe in a structured way what the control logic does while striving not to describe how the control logic is computed or implemented. By what we mean describing the sub-behaviors (called rules) of the control logic and by how we mean describing the lower-level implementation details (usually presented as states, transitions, encodings and other details of a FSA controller).

This choice to separate, as far as possible, what is computed from how it is computed has been especially made in the grammar-based approaches for the following reasons. Firstly, when the complexity of the control logic increases, describing the states and transitions of a FSA controller implementing the control logic becomes problematic. FSA controller of even a few states can have a large number of transitions and if some modifications should be made in the control logic, the FSA can change considerably. Secondly, the lower-level specifying how things are computed can be synthesized from the high-level control specification.

Typically, the synthesis begins with the construction of an abstract representation of the design (Data-Dependency Graph in our case) and then a translation (or transformation) is performed to obtain an initial FSA representation. In our case, and as in Attribute Grammars, we look to have a data/attribute evaluator (which consists of a set of DFA algorithms) rather than a FSA controller. The advantage of a data evaluator comes from the fact that not only one but multiple implementations of the control logic can be synthesized by analyzing the order of data evaluation (incremental, partial, total, parallel, etc.).

Thus, our method is based on describing the sub-behaviors of the control logic as a set of rules. The total behavior of a design is described by composing together the rules using compositional operators. Each rule links one input event to some output events (see Definition 6). When an input event is received, a rule will respond to this by executing computations, changing values of its attributes or sending output events. In a rule, the input event is linked to output events by a transition labeled by optional guard conditions. The guard conditions indicate the circumstances under which a rule can be applied.

To keep the rule definition simple, we define first input and output event.

Definition 4 (Input Event). *An input event v of a component* $C =< A, I, Imp, m >$ *is an element of* $(S_{in} \cup S_{int})$. □

Definition 5 (Output Event). *An output event v of a component* $C =< A, I, Imp, m >$ *is an element of* $(S_{out} \cup S_{int})$. □

Based on these events, a rule may specify four kinds of events (asynchronous events): receiving an input service, receiving an internal service, emitting an output service and emitting

an internal service. Table 5.5 gives some examples (with abbreviations) of such events.

Input Event → Output Events	Informal meaning
$s_1(arg_{s_1})[Guards] \rightarrow ...$	receipt of a service $s_1(arg_{s_1})$, which is an input or internal service.
$... \rightarrow s_2\$$	emission of a response $s_2\$$ of a service s_2, which is an input or internal service.
$... \rightarrow s_3(arg_{s_3})$	emission of a service $s_3(arg_{s_3})$, which is an output or internal service.
$s_4\$[Guards] \rightarrow ...$	receipt of a response $s_4\$$ of a service s_4, which is an output or internal service.

Table 5.5: Asynchronous events.

To take into account the synchronous events, we introduce a synchronization (a rendez-vous) symbol $\uparrow$. Thus, when a service is called, the caller waits until the service response returns. We describe this kind of event in table 5.6.

Input Event → Output Events	Informal meaning
$... \rightarrow s_1(arg_{s_1}) \uparrow$	emission of a service $s_1(arg_{s_1})$, and waiting for its result.

Table 5.6: Synchronous event.

In a rule r, we distinguish three types of data grouped as input, computed and output data. The input data denote the known data used during the computation achieved by the method implementing the service corresponding to the input event of r (this method is called F and it is defined hereafter in Definition 6). The input data consist only of internal component attributes and the arguments or result of the service causing the input event. The computed data consist of the results of F and the output data consist of the arguments or result of the service causing the output event. The output data are presented as the union of the input and computed data.

Guard conditions act on the input data. They ensure that the input data are valid or conforms to the conditions before applying the rule. They can be used, for instance, to ensure that two events are mutually exclusive if they occur at the same time.

Formally, a rule is defined as follows:

Definition 6 (Rule). *A rule describes the behavior of a component C when it receives an input event v. A rule is defined by a 4-tuple* $r =< L, Guards, R, E >$, *where:*

- $L = \{ v \}$ *with v is an input event. L represents the left side of the rule;*
- *Guards are the guard conditions, indicating the circumstances under which the input event v can be executed. A guard condition consists on a set of Boolean expressions. An input event v is executed if each Boolean expression is true;*
- $R = \{ v_1, ..., v_n \mid \forall i \in 1..n, v_i$ *is an output event* $\} \cup \{\emptyset\}$. *R represents the right side of the rule;*
- *E is a semantic equation which has the following form :*

$$(b_0, ..., b_q) = F(a_0, ..., a_p) \tag{5.1}$$

where F is a method that implements the service corresponding to the input event v and defines the computation of the output data (b_i) *in terms of the input data* (a_i). □

Before giving the definition of the constraints on the equation E, we define first three sets of data: Input Data ID_r, Computed Data CD_r and Output Data OD_r.

Definition 7 (Input data ID_r of a rule r). *Let a rule* $r =< L, Guards, R, E >$ *describes the behavior of a component* $C =< A, I, Imp, m >$ *when it receives an input event v, the input data ID of r are:*

$$v \in L, ID_r = \begin{cases} arg_s \cup A & \text{if} \quad v = s(arg_s) \\ \{s\$\} \cup A & \text{if} \quad v = s\$ \end{cases} \tag{5.2}$$

□

Definition 8 (Computed data CD_r of a rule r). *Let a rule* $r =< L, Guards, R, E >$, *computed data CD of r are the set of data resulting from the equation E:*

$$CD_r = \{ b_0, ..., b_q \} \tag{5.3}$$

□

Definition 9 (Output data OD_r of a rule r)**.** *Let a rule $r =< L, Guards, R, E >$, output data OD of r are the data emitted by the output events of r:*

$$OD_r = \bigcup_{v_i \in R} \begin{cases} arg_s & if \quad v_i = s(arg_s) \\ \{s\$\} & if \quad v_i = s\$ \end{cases} \tag{5.4}$$

□

Once these three sets of data are defined, the constraints on the semantic equation E of a rule r can be defined as follows:

Definition 10 (Constraints of a semantic equation)**.** *The constraints to be satisfied by a semantic equation $E : (b_0, ..., b_q) = F(a_0, ..., a_p)$ of a rule r are:*

- *Contraint (1): OD_r elements can only be elements of the union of ID_r and CD_r:*

$$OD_r \subseteq ID_r \cup CD_r \tag{5.5}$$

- *Contraint (2): F only accepts ID_r elements as inputs:*

$$\forall i \in 0..p, \ a_i \in ID_r \tag{5.6}$$

□

Example 3 (A behavior of the gossip component *Node*)**.** The following rule r indicates that the component *Node* receives the data $buffer_y$ from the outside through the service *gossip* and then responses by sending the data $buffer_x$ through the service *answer* if the condition *pull* is satisfied (*pull == True*):

$$r : \quad gossip(buffer_y), [pull] \quad \rightarrow answer(buffer_x) \quad F_r$$

In this rule, if the information has to be pulled (Boolean attribute *pull* is true), then the method F_r is executed. With the implementation of F_r is as follows:

$$
\begin{aligned}
(buffer_x) = F_r(buffer_y)\{ \\
&buffer = (MyAddress, 0) \\
&view.permute() \\
&view.moveOldestH() \\
&buffer.append(view.head(c/2-1)) \\
&buffer_x = buffer \\
&answer(buffer_x) \\
&view.select(c, H, S, buffer_y) \\
&view.increaseAge() \\
\}
\end{aligned}
$$

During the execution of F_r, a *buffer* is initialized with a fresh information. Then, $c/2 - 1$ elements are appended to the *buffer*. These elements are selected randomly from the *view* without replacement, ignoring the oldest H elements. The *buffer* created this way is sent through the service *answer*. Then, the received $buffer_y$ is passed to procedure $select(c, H, S, buffer_y)$, which creates the new view based on the listed parameters and the current view. Finally, the view is updated with new age.

The input data ID_r of this rule are the union of the attributes $\{view,\ c,\ buffer,\ H,\ S\}$ and the argument $buffer_y$ of the input service *gossip*:

$$ID_r = \{view,\ c,\ pull,\ H,\ S\} \cup \{buffer_y\}$$

The computed data CD_r are the data resulting from the execution of the method F_r:

$$CD_r = \{buffer_x, view\}$$

The output data OD_r consist of the arguments of the emitted service *answer*:

$$OD_r = \{buffer_x\}$$ □

In right side R of a rule, output events (separated by ";") may be output service emitted to different remote components, and each component is a process that can be executed separately. This parallel relation between output events is nearly implicit. For example, $r : s \rightarrow s_1, s_2$ means services s_1 and s_2 do not have sequential relation.

This relation characterizes the activity of a unique rule. So, in order to characterize the activity of a set of rules, we define three operations for rules:

- *Sequence operation* " ; ": Indicating a sequential order among rules. For example, $r_1; r_2; r_3$ means rule r_1 acts before r_2 and r_2 acts before r_3.
- *Alternative operation* " | ": Indicating an alternative choice concerning the output events of a rule. For example,

$$\begin{aligned} r : s[Guards] \rightarrow \quad & s_1 \\ & \mid s_2 \end{aligned}$$

means services s_1 and s_2 may have same chance to occur. This alternative can be controlled by the guard conditions.

- *Recursive operation* "[ ]": Indicating that an internal service s will be called recursively. This recursion can be controlled by the guard conditions. Thus, recursion operations can be used to have repetition (loop) indicating that some rules will be executed n times continuously. For example,

$$\begin{aligned} [r_1 : \quad & s[Guards] \quad \rightarrow s_1 \\ r_2 : \quad & s_1\$ \qquad\quad \rightarrow s] \end{aligned}$$

means that the rule r_1 execute the internal service s if guard conditions are satisfied, and then it calls the service s_1. When the service s_1 response arrives, the rule r_2 calls the internal service s, which will be executed again by r_1 if guard conditions are still satisfied.

Therefore, from the definition of an interface, a rule and rule operations, we have the following definition of a component behavior.

Definition 11 (Behavior)**.** *The behavior of a component C is a set of rules combined by sequence, alternative and recursion operations with respect to the following regular expressions:*

$$B ::= r^+ \mid [B^+] \mid \{B^+\} \tag{5.7}$$

$$r ::= r \mid (r \backslash r) \tag{5.8}$$

□

Example 4 (The behavior of the gossip component *Node*)**.** According to definition 11, the behavior of the gossip component *Node* is $B_{Node} = \{r_1, r_2, r_3, r_4\}$, where:

$$
\begin{array}{lll}
r_1: & timeout(T) & \rightarrow gossip(buffer_{in}) \quad F_{r_1} \\
r_2: & answer(buffer_{out}), [pull] & \rightarrow \quad F_{r_2} \\
r_3: & gossip(buffer_{out}), [pull] & \rightarrow answer(buffer_{in}), updateView(buffer_{out}) \quad F_{r_3} \\
r_4: & updateView(buffer_{out}) & \rightarrow \quad F_{r_4}
\end{array}
$$

□

5.3.4 System

The component composition is based on connections among component instances. A connection between two instances occurs when one of them provides its interface and another instance uses it. Hence, input (resp. output) services are connected to signature-matching output (resp. input) services. There is a unique connection between two instances.

Once component instances are connected, the behavior of the entire resulting system is obtained by composition of behaviors of participating instances. Since the component instance behavior is a set of rules connected by sequence, alternative and recursive operations, the system behavior can be again viewed as a set of rules connected by these same operations.

Formally, a system is defined as follows:

Definition 12 (System)**.** *A system is defined by a 2-tuple* $Sys =< Inst, Con >$ *where:*

- $Inst$ *is a set of component instances;*
- $Con = \{ (c_1, c_2) \mid (c_1, c_2) \in Inst \times Inst \}$ *is a set of connections between instances.* □

Example 5 (A gossip system)**.** According to Definition 12, a gossip system constituted of two instances ($node_x$ and $node_y$) is expressed as $GossipSys = < Inst, Con >$, where:

- $Inst = \{node_x : Node, node_y : Node\}$;
- $Con = \{(node_x, node_y)\}$.

Now, we define the system behavior from the behavior of each underlying component instance. To achieve this, we associate the source and the destination instances to the events of the rules. For example, let a rule $r : v \rightarrow v_1, v_2$ describes the behavior of a component C when

receiving the input event v, where $v \in S_{in}$ and $S_{in} \in I_C$, and let connections (c, c_i) and (c, c_j), where c, c_i and c_j are instances of, respectively, C, C_i and C_j components. The rule r will be transformed to $(v, c_i) \rightarrow (v_1, c_j), (v_2, c_i)$ if the source instance causing the input event v is c_i and the destination instances of the output events v_1 and v_2 are C_j and C_i, respectively. This transformation is performed by a function, which specifies in each rule the source component instance causing its input event and the destination component instances of its output events.

Definition 13 (Rule Transformation). *Let a rule $r =< L, Guards, R, E >$ describes the behavior of a component C_i when it receives a input event $v \in L$, and let a connection $(c_i, c_j) \in Con$, where c_i and c_j are instances of, respectively, C_i and C_j components. The transformation of r when c_i is connected to c_j is $r^{c_i} = \sigma(r)_{/c_i \rightarrow c_j}$, where:*

$$\begin{aligned} \sigma(r)_{/c_i \rightarrow c_j} &= \sigma(r : v \rightarrow v_1 ... v_n)_{/c_i \rightarrow c_j} \\ &= r : \sigma_r(v)_{/c_i \rightarrow c_j} \rightarrow \sigma_r(v_1)_{/c_i \rightarrow c_j} ... \sigma_r(v_n)_{/c_i \rightarrow c_j} \end{aligned} \tag{5.9}$$

with the transformation function σ is defined as follows:

$$\sigma \; : \; V \overset{r,/c_i \rightarrow c_j}{\longrightarrow} V \times Inst \; or \; V$$

$$\sigma_r(v)_{/c_i \rightarrow c_j} = \begin{cases} (v, c_j) & \text{if} \quad v \in r.L \wedge v \in S_{in}(c_i) \cap S_{out}(c_j) \\ (v, c_j) & \text{if} \quad v \in r.R \wedge v \in S_{out}(c_i) \cap S_{in}(c_j) \\ v & \end{cases} \tag{5.10}$$

□

Therefore, the system behavior is defined as follows:

Definition 14 (System Behavior). *A system behavior $B(Sys)$ is a set of rules combined by sequence, alternative and recursion operations, where:*

$$B(Sys) = \bigcup_{(c_i, c_j) \in Con} \left\{ B(c_i)_{/c_i \rightarrow c_j} \cup B(c_j)_{/c_j \rightarrow c_i} \right\} \tag{5.11}$$

$$B(c_x)_{/c_x \rightarrow c_y} = \{ \sigma(r)_{/c_x \rightarrow c_y} | r \in B(c_x) \wedge r.L \in S_{in}(c_x) \cap S_{out}(c_y) \} \tag{5.12}$$

□

Example 6. According to Definition 14, the behavior of the gossip system $GossipSys = < Inst, Con >$ (presented in example 5) is expressed as $B(GossipSys)$, where:

- $B(GossipSys) = B(node_x)_{/node_x \to node_y} \cup B(node_y)_{/node_y \to node_x}$
- $B(node_x)_{/node_x \to node_y} = \{r_1^x, r_2^x\}$
- $B(node_y)_{/node_y \to node_x} = \{r_3^y, r_4^y\}$

With r_1^x, r_2^x, r_3^y and r_4^y are specified as follow:

Gossip System behavior	**Hidden implementations**
Server activity ($node_x$)	
$r_1^x : timeout(T) \rightarrow (gossip(buffer_x), node_y)$	$(buffer_x) = F_{r_1}()\{$ $p = view.selectPeer()$ **if**$(push)$ $buffer_x = (myAddress, 0)$ $view.permute()$ $view.moveOldestH()$ $buffer_x.append(view.head(c/2-1))$ $gossip(buffer_x)$ **else** $buffer_x = null$ $gossip(buffer_x)$ $view.increaseAge()$ $\}$
$r_2^x : (answer(buffer_y), node_y) \rightarrow$	$F_{r_2}(buffer_y)\{$ **if**$(pull)$ $view.select(c, H, S, buffer_y)$ $\}$
Customer activity ($node_y$)	
$r_3^y : (gossip(buffer_x), node_x), [pull] \rightarrow$ $(answer(buffer_y), Node_x)$	$(buffer_y) = F_{r_3}(buffer_x)\{$ $buffer_y = (MyAddress, 0)$ $view.permute()$ $view.moveOldestH()$ $buffer_y.append(view.head(c/2-1))$ $answer(buffer_y)$ $\}$
$r_4^y : (gossip(buffer_x), node_x), [\neg pull] \rightarrow$	$F_{r_4}(buffer_x)\{$ $view.select(c, H, S, buffer_x)$ $view.increaseAge()$ $\}$

Table 5.7: Behavior of a *Gossip System* constituted of two nodes ($node_x$ and $node_y$).

5.4 Defining a simple generic P2P system

The idea of P2P is applied in various contexts, and P2P systems do not necessarily have several characteristics in common; neither do they have to rely on a fixed set of attributes. There are no major standardization initiatives that look at all aspects of P2P technology and computing. The term P2P is defined by its usage and unique formal definition of P2P computing does not exist [Mauthe and Hutchison, 2003]. However, there are a number of features many P2P systems share as introduced in the following well-known and academically accepted definitions:

- The Gartner Group [Gartner Research Group, 2001] defines P2P computing as: *"characterized by direct connections using virtual namespaces, it describes a set of computing nodes that treat each other as equals (peers) and supply processing power, content or applications to other nodes in a distributed manner, with no presumptions about a hierarchy of control".*
- A brief concise definition of P2P computing is given in [Hofmann and Beaumont, 2005]: *"a set of technologies that enable the direct exchange of services or data between computers".*
- A more recent definition is given in [Taylor et al., 2009]: *"The peer-to-peer (P2P) architectural style consists of a network of loosely coupled autonomous peers, each peer acting both as a client and a server. Peers communicate using a network protocol, sometimes specialized for P2P communication-such was the case for the original Napster and Gnutella file-sharing applications. Unlike the client-server style where state and logic are centralized on the server, P2P decentralizes both information and control."*

These definitions highlight the following elements that are fundamental to P2P computing and common in describing P2P applications: i) direct exchange of resources between peers; ii) each peer is independent and equivalent in functions; iii) there are no center servers or controllers; iv) peers communicate using a network protocol.

In addition to these elements, we adopt the position proposed in [Barkai, 2002]: *"One way to derive a definition of purpose that is more inclusive, flexible, and extensible is this: There are P2P technologies, and there is P2P computing".* The P2P technologies allow peers to share

resources and collaborate on computational tasks. This implies an abundance of supporting technologies, such as discovery, remote resource management, security and more. P2P computing is the use of P2P technologies. A resulting phenomenon is the creation of an overlay community (of peers/components) that collaborates through resource (data, services, ...) sharing. This is the immediate result and operational purpose of P2P computing.

As can be understood from the above definitions, the P2P system we define with DDF is formed by establishing an overlay network between peers. Peers are represented by component instances. The same notation is used to refer to the component instances as well as the peers they represent. Each component instance acts both as a server (with its input services) and a client (with its output services). Each component instance is used to store resources (data) which are accessible through services. Each instance is connected to a bounded number of other instances and has a unique identifier, such as an IP address. As the network evolves, instances can continuously seek after new partners by implementing a specific algorithm such as Gossip algorithm (specified in Section 5.2). Thus, the final structure of the P2P network depends on the kinds of these searching algorithms. We assume the existence of an underlying layer (SON infrastructure in our case) that provides to component instances the necessary lookup service (like a DHT; cf. Section 7.3.2) and communication mechanisms (like JXTA; cf. Section 7.3.3). These assumptions allow us to make only very weak networking issues at the high level description and defer the additional ones to the lowest level where they are needed. Thus, we provide a simple generic definition that can be implemented in different environments with different low level assumptions.

Formally, we have defined a P2P system by extending Definition 12, and while trying to get closer to the one proposed by [Giesecke et al., 2005] because it was specified to be easily augmented by other P2P functionalities. Thus, our definition of a P2P system is as follows:

Definition 15 (P2P system)**.** *A P2P system is defined by a 4-tuple* $P2PSys =< Inst, Con, \gamma, \delta >$*:*

- $Inst$ *is a set of component instances;*
- $Con = \{(c_1, c_2) | (c_1, c_2) \in Inst \times Inst\}$ *is a set of connections between component instances;*
- $\gamma : Inst \rightarrow ID$ *is a mapping function that maps each component instance* $c \in Inst$ *to its*

identifier $id \in ID$, where ID is a set of identifier;

- $\delta : T \rightarrow Inst_T \times Con_T$ *is a time mapping function that maps an instant $t \in T$ to the set of instances and connections that are available in this instant.* □

The time mapping function describes the evolution of the system over time. For example, if a component instance c (a peer) joins the system at time t_i and leaves at t_j, then c is in the image of $t \in [t_i, t_j[$. During the time interval $[t_i, t_j[$, c may open and close many connections to already connected instances. These connections are also captured by the time mapping function and are in the image of $t \in [t_i, t_j[$.

The time mapping function δ may control several P2P features (e.g., reach a certain position in the network topology, adapt to failures, etc.), but the way how it is incorporated into the specification depends on the protocol implemented by the network peers. For instance, in Gossip protocol, each peer maintains a local table $view(ID, age)$ providing a partial view on the complete set of memberships and periodically refreshes the table using a gossiping procedure.

To illustrate the evolution of a P2P system over time, we use an example similar to the one proposed in [Giesecke et al., 2005]. The system consists of seven peers and ten connections. For all instants $t \in [t_0, t_1[$ the peers c_1 to c_5 participate in the system (see Figure 5.5). c_5 leaves the system at t_1 and, therefore, its connections disappear as well. c_4 had a connection to c_5 and could ask c_3 about other peers. c_3 could send the ip_{c_1} to c_4, so that c_4 opens a connection to c_1 at instant t_2. At the same time, new two peers c_6 and c_7 enters the system with a connection to c_3 and c_2, respectively. After t_3, c_3 mediates connections to c_1. Formally, this example is defined as follows:

$Inst = \{c_1, c_2, c_3, c_4, c_5, c_6, c_7\}$;

$Con = \{con_1, con_2, con_3, con_4, con_5, con_6, con_7, con_8, con_9, con_{10}\}$.

$ID = \{ip_{c_1}, ip_{c_2}, ip_{c_3}, ip_{c_4}, ip_{c_5}, ip_{c_6}, ip_{c_7}, \}$;

Con	con_1	con_2	con_3	con_4	con_5	con_6	con_7	con_8	con_9	con_{10}
(c_i, c_j)	(c_1, c_2)	(c_2, c_3)	(c_1, c_3)	(c_3, c_4)	(c_3, c_5)	(c_5, c_6)	(c_1, c_4)	(c_3, c_6)	(c_2, c_7)	(c_1, c_6)

γ :

$Inst$	c_1	c_2	c_3	c_4	c_5	c_6	c_7
ID	ip_{c_1}	ip_{c_2}	ip_{c_3}	ip_{c_4}	ip_{c_5}	ip_{c_6}	ip_{c_7}

δ :

T	$Inst_T$	Con_T
$[t_0, t_1[$	c_1, c_2, c_3, c_4, c_5	$con_1, con_2, con_3, con_4, con_5, con_6$
$[t_1, t_2[$	c_1, c_2, c_3, c_4	$con_1, con_2, con_3, con_4$
$[t_2, t_3[$	$c_1, c_2, c_3, c_4, c_6, c_7$	$con_1, con_2, con_3, con_4, con_7, con_8, con_9$
$[t_3, t_4[$	$c_1, c_2, c_3, c_4, c_6, c_7$	$con_1, con_2, con_3, con_4, con_7, con_8, con_9, con_{10}$

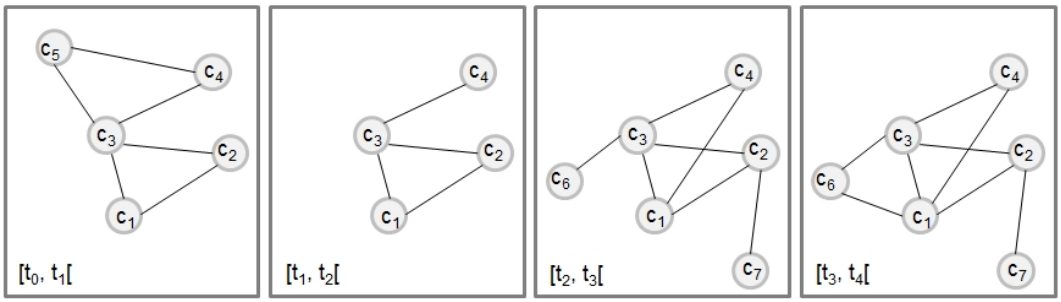

Figure 5.5: Illustration of an evolution of a P2P system.

Chapter 6

Analysis of DDF specification

Contents

This chapter aims at presenting how we analyze application behavior specified with our Data Dependency Formalism, presented in the previous chapter. The first step of this analysis is to construct a Data-Dependency Graph (DDG) that we introduce in Section 6.1 and 6.2. After that, verifying a property is reduced to find a solution to a set of constraints (called data-flow equations) on the inputs and the outputs of the graph nodes. This is illustrated through two examples presented in Section 6.3.1 and 6.3.2. The first example consists of checking the property of deadlock freedom which is reduced to find whether a node in the graph depends on itself. The second example is about dominance property that has many applications in computer science (code optimization, detection of parallelism, construct of hierarchical overlay networks, optimizing routing, memory usage analysis, etc.). To compute dominance information in a DDG, we formulate the problem as a set of data-flow equations that defines a set of dominators for each graph node. These equations are solved with an iterative algorithm.

6.1 Introduction

As described in Section 2.2.2.2, Data-Flow Analysis refers to a body of techniques, which derive information about the flow of data along program execution paths in order to infer or compute some properties. To achieve this, we must first consider all the possible paths through a flow graph that the program can take. Therefore, we have defined a Data-Dependency Graph (DDG). It presents an abstract representation of a system. This abstraction exposes the right level of detail to perform DFA.

The DDG models the flow of data values from the point where a data value is created, a definition, to any point in a configuration where it is used, a use. A node in a DDG represents a low-level operation on data. In most cases, nodes contain both definitions and uses. A directed edge in a DDG connects two nodes (head and tail). The head defines a data value and the tail uses it. The edges in the DDG represent interesting constraints on the control flow, i.e., a data value can be used only if it has been defined. This only implies a partial order on the execution. Therefore, no total order among configuration operations is needed to be given by the designer who often set it as an automaton to perform analysis. Moreover, it is possible through a data-flow analysis on this graph to infer various data evaluation orders during run time (e.g., total, parallel and incremental). Thanks to the theory of iterative data-flow analysis based on a fixed-point theorem [Kam and Ullman, 1976].

6.2 Data-Dependency Graph

The Data-Dependency Graph is extracted from the semantic equations of the system by connecting together the Rule-Dependency Graphs corresponding to each rule used in this system. The Rule-Dependency Graph of a rule r describes internal and external dependency relations of input and output data, which are manipulated by the different services of r.

The internal relations are induced from the semantic equation of a given rule, which define the computation of the output data in terms of the input data. Thus, Definition 16 defines the internal dependency relation as follows:

Definition 16 (Internal Dependency Relation)**.** *The internal dependency relation* $G_{int}(r)$ *in* $ID_r \times OD_r$ *of a rule r is defined as follows:*

$$a_p \; G_{int}(r) \; a_q \quad \textit{if and only if} \quad (..., a_q, ...) = F(..., a_p, ...) \tag{6.1}$$

□

Thus, a_q **depends on** a_p, if a_p is an argument in the semantic equation for a_q.

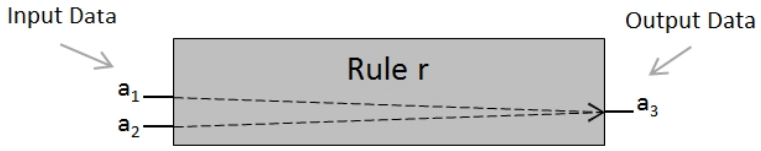

Figure 6.1: Example of an internal dependency relation.

Figure 6.1 shows an example of an internal dependency relation where output data a_3 depends on input data a_1 and a_2.

The external relations of a rule *r* are related to the source and destinations of events present in *r*. Therefore, we present the definition of external dependency relation as follows:

Definition 17 (External Dependency Relation)**.** *Let a rule* r^e *describes the behavior of a component instance e when it receives an input event, and let* $(v(a_1^e, ..., a_q^e), e')$ *be an event in* r^e*. The external relations induced from the event* $(v(a_1^e, ..., a_q^e), e')$ *depend on the position of this event in* r^e*:*

$$if \; (v(a_1^e, ..., a_q^e), e') \in r^e.L \quad then \quad \forall k \in 1..q, \quad a_k^{e'} \; G_{ext}(r) \; a_k^e \tag{6.2}$$

$$if \; (v(a_1^e, ..., a_q^e), e') \in r^e.R \quad then \quad \forall k \in 1..q, \quad a_k^e \; G_{ext}(r) \; a_k^{e'} \tag{6.3}$$

□

Thus, a_k^e **depends on** $a_k^{e'}$, if a_k^e is an argument in the input event received from e'. And $a_k^{e'}$ **depends on** a_k^e, if $a_k^{e'}$ is an argument in the output event emitted to e'.

Figure 6.2 shows an example of an external dependency relation where data of an input event in rule r^e depend on data which are output in $r^{e'}$.

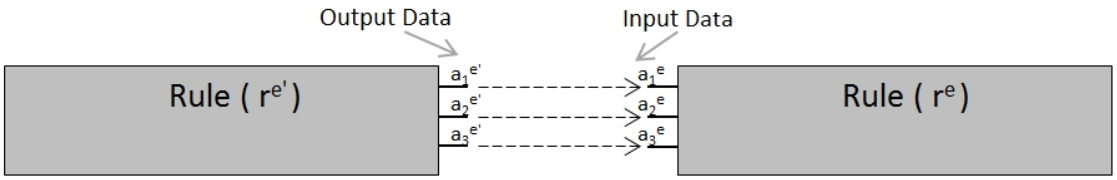

Figure 6.2: Example of an external dependency relation.

When no confusion arises between the notions of relation and graph, we shall represent them both by the same notation. Accordingly, we denote the Internal Dependency Graph of a rule $G_{int}(r)$ and the External Dependency Graph $G_{ext}(r)$. The union of these two graphs represents the Rule-Dependency Graphs of r, which we denoted by $G(r)$.

The Data-Dependency Graph, the graph of the whole system, is obtained from the union of the Rule-Dependency Graphs and it is defined as follows:

Definition 18 (Data-Dependency Graph)**.** *Let* $Sys =< Inst, Con >$ *be a system, the Data-Dependency Graph of the system Sys is:*

$$G(Sys) = \bigcup_{e \in Inst} \left(\bigcup_{r \in B(e)} (G_{int}(r) \cup G_{ext}(r)) \right) \tag{6.4}$$

□

Example 7. To illustrate how to construct a simple Data-Dependency Graph, we consider the push-pull version (boolean attributes *push* and *pull* are true) of the system *GossipSys* presented in Example 5. As specified before, the behavior of *GossipSys* is $B(GossipSys) = \{r_1^x, r_2^x, r_3^x, r_4^y\}$, where:

$$\begin{array}{lll}
r_1^x: & timeout(t) & \rightarrow (gossip(buffer_x), Node_y) \quad F_{r_1^x} \\
r_2^x: & (answer(buffer_y), Node_y) & \rightarrow \quad F_{r_2^x} \\
r_3^y: & (gossip(buffer_x), Node_x) & \rightarrow (answer(buffer_y), Node_x), updateView(buffer_x) \quad F_{r_3^y} \\
r_4^y: & updateView(buffer_x) & \rightarrow \quad F_{r_4^y}
\end{array}$$

The Data-Dependency Graph of *GossipSys* at the end of *timeout*(t) is shown in Figure 6.7. This graph is obtained from the union of the Internal Dependency Graphs of the rules r_1^x, r_2^x, r_3^x and r_4^y, which are presented in Figures 6.3, 6.4, 6.5 and 6.6, respectively.

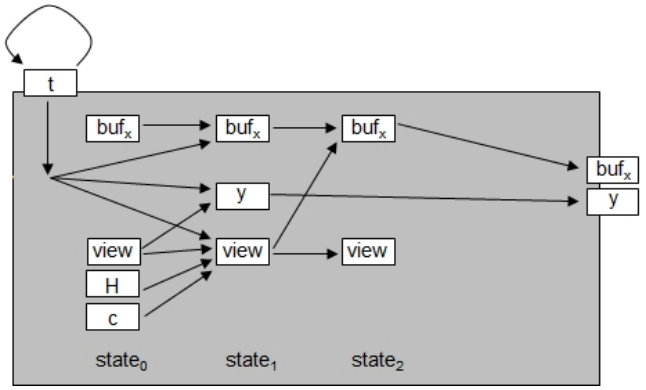

Figure 6.3: Internal Dependency Graph of the rule r_1^x.

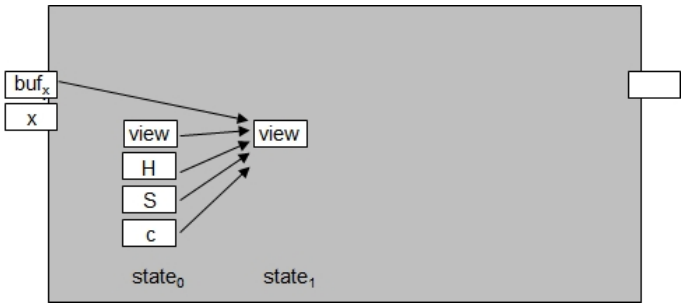

Figure 6.4: Internal Dependency Graph of the rule r_2^x.

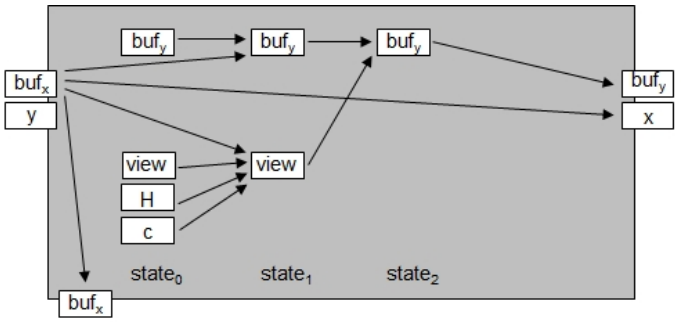

Figure 6.5: Internal Dependency Graph of the rule r_3^x.

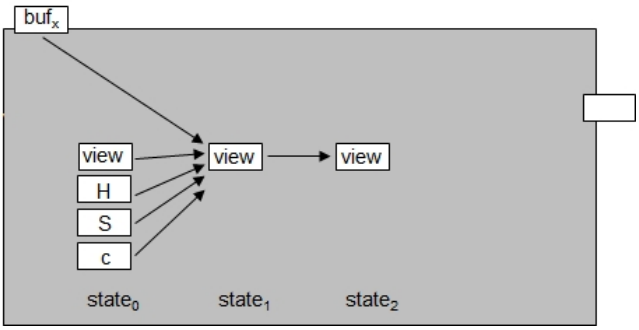

Figure 6.6: Internal Dependency Graph of the rule r_4^x.

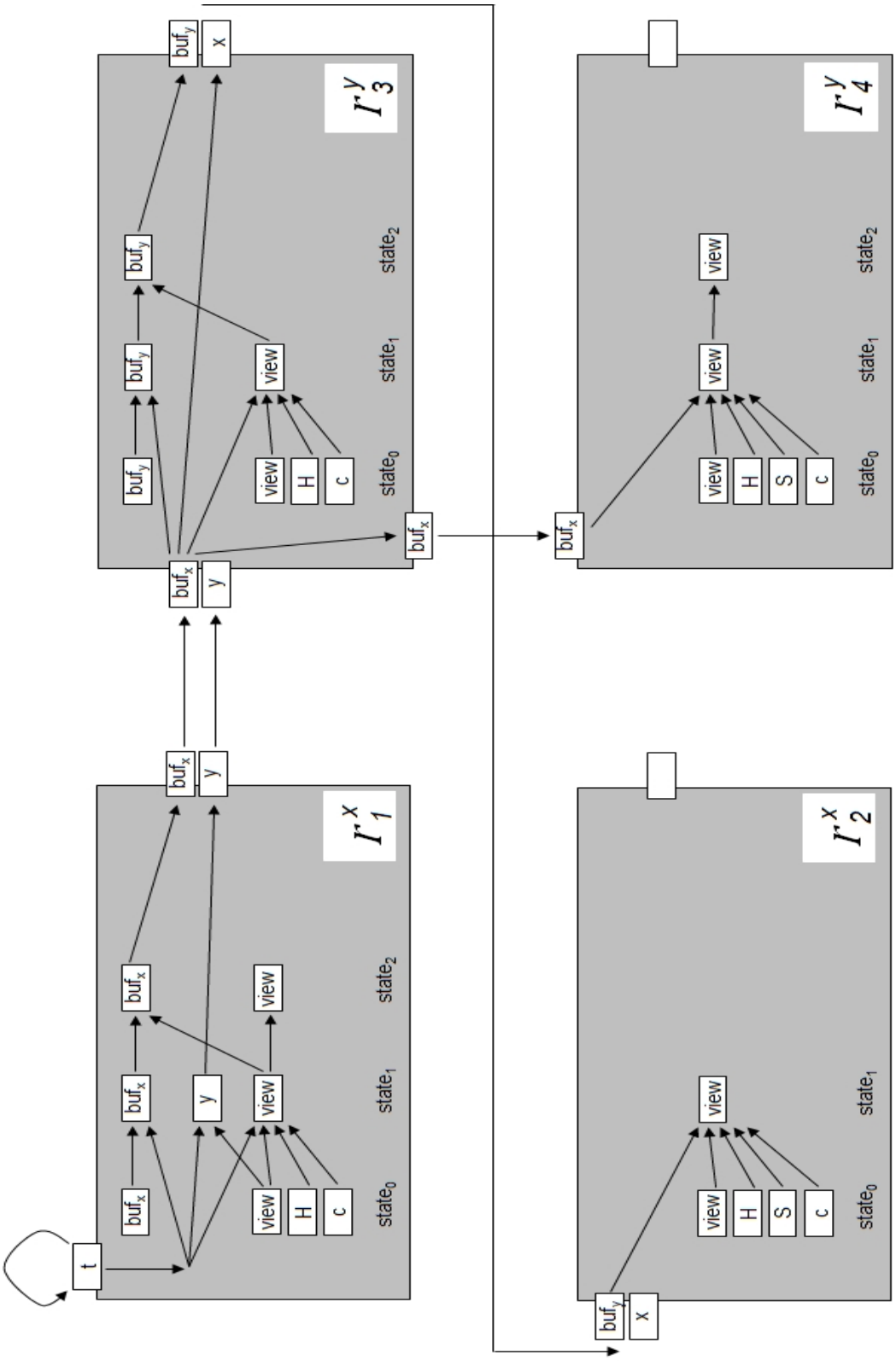

Figure 6.7: An example of a Data-Dependency Graph.

6.3 Analysis examples

6.3.1 Detection of deadlocks

In a component composition, services are often forced to wait for resources from other services to finish execution. If the resources are not available, then the system may enter an infinite wait state. Under the assumption that this issue is not caused by infinite loops, infinite wait is always caused by deadlocks or starvations. A deadlock is a situation in which two or more actions (services) are mutually waiting on each other to finish, while a starvation is a situation in which an action is perpetually denied access to resources needed to make progress.

A system deadlock can be viewed as a circular dependency between data exchanged through services. Therefore, the basis of our deadlock analysis is detecting possible circular dependencies in the Data-Dependency Graph of the system.

Once the DDG is defined, we can induce if the system is deadlocked or not by searching for circularity in the graph. In other words, we shall search for a datum which depends on itself. An example of such as situation is given in Figure 6.8.

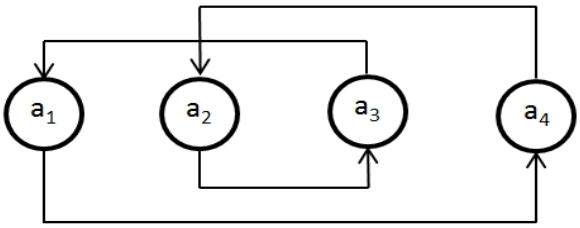

Figure 6.8: Example of data which depend on themselves.

Formally, a deadlocked system is defined as follows:

Definition 19 (Deadlocked system)**.** *Let* $Sys =< Inst, Con >$ *be a system and* $G(Sys) = \cup_{e \in Inst}(\cup_{r \in B(e)}(G(r)))$ *be the Data-Dependency Graph of Sys. Then Sys is said to be deadlocked if and only if there exist a rule* $r \in B(e)$, $e \in Inst$ *such that* $G(r)$ *contains a cycle.* □

Now, we present an algorithm (Algorithm 3) which determines whether or not a system is deadlocked. The first stage of our deadlock test algorithm is to construct the Rule-Dependency Graph $G(r)$ of each rule r in the behavior of each component in the system. This construction

is achieved by connecting together the internal and external dependency graph of r obtained as described above. After that, $G(r)$ is added to the Data-Dependency Graph $G(Sys)$ which is initially empty. Once all rule graphs are added to $G(Sys)$, we compute transitive closure of $G(Sys)$, which we denoted by $G(Sys)^+$, in order to add induced dependencies. Those induced dependencies allow us to determine whether or not a node (a datum) of the graph is circular. If this is true, then we deduce that the system has a deadlock and a message with the rule that contains the circular data is printed.

Algorithm 3 Deadlock test

Require: $Sys =< Inst, Con >; G(Sys) := \emptyset;$

```
{ - - - - - - - - - - Construction of the system graph - - - - - - - - - -}
for all e ∈ Inst do
   for all r ∈ B(e) such that r : (v_0, e_0) → (v_1, e_1), ..., (v_n, e_n) do
      G(r) := G_int(r) ∪ G_ext(r);
      G(Sys) := G(Sys) ∪ G(r);
   end for
end for

{ - - - - - - - - - - - - - - Search for deadlocks - - - - - - - - - - - - - -}
G(Sys) := G(Sys)^+;
for all e ∈ E do
   for all r ∈ B(e) such that r : (v_0, e_0) → (v_1, e_1), ..., (v_n, e_n) do
      if G(Sys)_/r contains a cycle then
         print Deadlock detected in rule r;
      end if
   end for
end for
```

6.3.2 Dominance analysis

Dominance analysis is a concept from graph theory, and it has many applications not only in the real world, but also in computer science. In compilers, dominance analysis is mostly used in code optimization, and it is performed over flow graphs representing the execution of programs. One important task in this context is the optimization of loops since the execution of programs tends to spend most of their time in their inner loops. In parallel computing, dominance analysis is used to compute control dependences that identify those conditions affecting statement execution. Such information is critical for detection of parallelism [Srinivasan and Wolfe, 1992].

In peer-to-peer applications, dominance analysis can be used to construct hierarchical overlay networks for more efficient index searching. It can also be used for optimizing routing among a set of nodes by reducing the searching space for a route to the dominating nodes in the set. Dominating nodes are a small set of nodes which are close to all other. Another field where dominance analysis is applied is memory usage analysis. In this field, the dominator tree (defined hereafter) is used to easily find memory leaks and identify high memory usage.

In a Data Dependency Graph, we say that node d_i dominates node d_j, written $di\ dom\ dj$, if every path from the entry node of the graph to d_j goes through d_i.

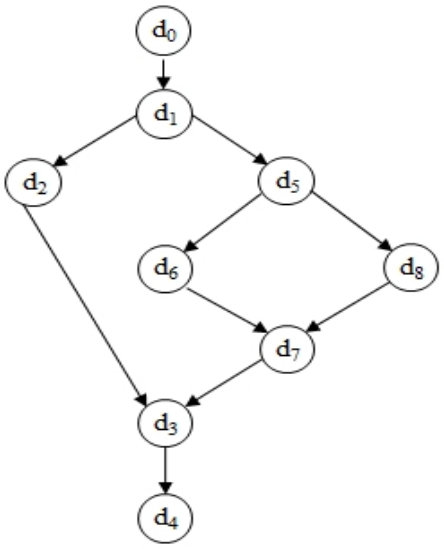

Figure 6.9: A Data Dependency Graph.

To make this dominance notion concrete, we consider the Data Dependency Graph of Figure 6.9. In this graph, the nodes d_0, d_1, d_5, and d_8 lie on every path from d_0 to d_8. Thus, $Dom(d_8)$ is the set $\{d0, d1, d5, d8\}$. The other sets of dominators for the graph are as follow:

$$
\begin{aligned}
Dom(d_0) &= \{d0\} \\
Dom(d_1) &= \{d_0, d_1\} \\
Dom(d_2) &= \{d_0, d_1, d_2\} \\
Dom(d_3) &= \{d_0, d_1, d_3\} \\
Dom(d_4) &= \{d_0, d_1, d_3, d_4\} \\
Dom(d_5) &= \{d_0, d_1, d_5\} \\
Dom(d_6) &= \{d_0, d_1, d_5, d_6\} \\
Dom(d_7) &= \{d_0, d_1, d_5, d_7\} \\
Dom(d_8) &= \{d_0, d_1, d_5, d_8\}
\end{aligned}
$$

A useful way of presenting dominance information is a dominator tree, in which each node

d dominates only its descendants [Aho et al., 1986]. For example, Figure 6.10 shows the dominator tree for the DDG of Figure 6.9. We note here that d_7 is a child of d_5, even though it is not an immediate successor of d_5 in the DDG. This is because that each node d_i in the tree has a unique immediate dominator d_j which is the last dominator of d_i in the DDG.

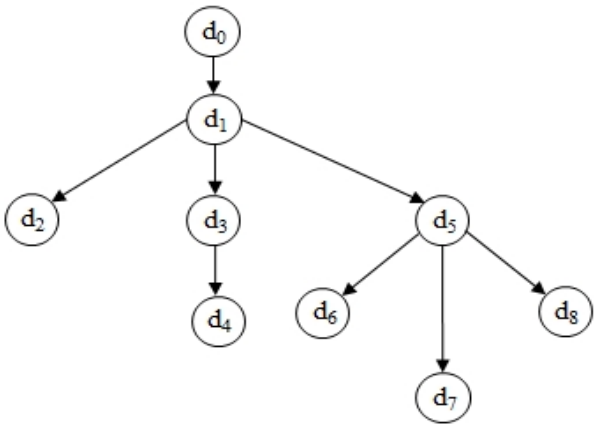

Figure 6.10: Dominator tree for the DDG of Figure 6.9.

To compute dominance information in a DDG, we can formulate the problem as a set of data-flow equations and solve them with an iterative algorithm. This algorithm is based on the one proposed by [Allen and Cocke, 1972] who relied on the principles of data-flow analysis to

Algorithm 4 Iterative Dominator Algorithm

Require: $G(Sys) := (N, E)$;

for all $n \in N$ **do**
  $Dom(n) = N$;
end for

$changed := True$;

while $changed$ **do**
  $changed := False$;
  for all $n \in N$ **do**
    $temp = \{n\} \cup (\bigcap_{m \in preds(n)} Dom(m))$;
    if $temp \neq Dom(n)$ **then**
      $Dom(n) = temp$;
      $changed := True$;
    end if
  end for
end while

guarantee termination and correctness.

Given a $DDG = (N, E)$, where N is a set of nodes and E is a set of directed edges, the Dom sets are defined by the following data-flow equations:

$$Dom(n) = \{n\} \cup \left(\bigcap_{m \in preds(n)} Dom(m) \right) \tag{6.5}$$

The initial conditions of the equations are: $Dom(n_0) = n_0$, and $\forall n \neq n_0$, $Dom(n) = N$. $preds$ is a relation, defined over E, that maps each node to its predecessors in the graph. Algorithm 4 shows an iterative solution for these dominance equations. It initializes the Dom set for each node, then repeatedly computes those sets until they stop changing.

Chapter 7

SON: A runtime middleware

Contents

This chapter describes fundamental aspects of the SON (Shared-data Overlay Network) middleware. SON is based on the concepts (i.e., component, service, interface, etc.) defined and formalized in our DDF for developing and deploying component-based P2P applications. This chapter addresses aspects referring to the structure of SON, its underlying component model and communication model. Besides those conceptual issues, the chapter presents a summary of the prototypical implementation and shows how SON middleware can be used to support the development of component-based P2P applications through two examples: SGT (Sim-

ple Georeferencing Tool) which is a lightweight application dedicated to collect, process and display georeferenced data, and P2Prec (a social based P2P recommendation system) which is developed in our research team for large-scale data sharing.

7.1 Overview

It is expected that the applications specified with our DDF formalism would need to run in distributed and ubiquitous environments. In this context, application components must be able to communicate with each other through the network. In addition, they must be able to adapt according to their evolution and execution environment. We say that the application (architecture) is dynamic [McKinley et al., 2004]. To meet these constraints, we adopted a service-oriented component approach to develop a middleware called SON (Shared-data Overlay Network), available online [SON, 2011] as an open-source software.

SON is based on the concepts defined in the DDF for developing and deploying applications in a simple and effective way. SON combines three powerful paradigms: CBSE, P2P and Service-Oriented Architecture (SOA) [Papazoglou and Heuvel, 2003]. As described in Section 3.2, SOA is a software architecture that uses services as fundamental elements for developing applications. SOA is based on three actors: i) the Service Provider publishes on a Service Broker the service descriptions which specify both the available service operations and how to invoke them (e.g., network protocol that must be used for the invocation, software components required to establish the connection, etc.); ii) the Service Broker registers the service descriptions and references; and iii) the Service Consumer discovers the services by running a search on the Service Broker. It then establishes a connection with the provider to invoke the service operations. The SOA design principles (cf. Section 3.2.2) allows the development of modular, loosely coupled and dynamic applications. Along this chapter, we show how these SOA design principles can be integrated into our middleware while being separated from the implementation code.

SON middleware is composed of a component model and a connection model (see Figure 7.1). The component model defines how to create and validate components. The connection

model provides not only local and distributed communication mechanisms, but also allows different peers to publish and search resources. In this context, a resource represents a component that provides or requires services, and a peer represents a set of locally interconnected components.

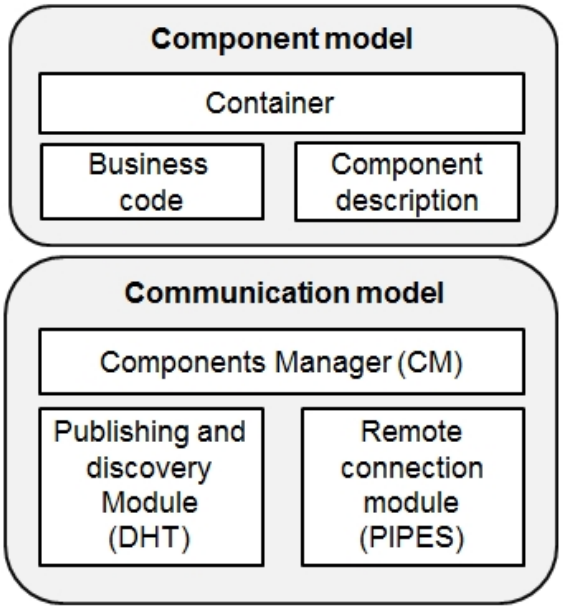

Figure 7.1: Overview of SON middleware.

By using the DDF specifications, SON's user is able not only to check the consistency between each DDF rule and the corresponding implementation, but also to perform an effective code generation, i.e., the target implementation and generated code fit well the behavioral constraints contained in the DDF rules. In fact, the user defines for each component a set of services (input, internal and output) and behavioral rules. Then, he implements the code of the components, i.e., the method *F* that implements the service corresponding to the input event of each rule (cf. Definition 6). Afterwards, a code generation tool, called Component Generator (CG), checks whether the implementation of each method *F* is valid and fits well the constraints (cf. Definition 10) contained in the corresponding DDF rule. Once the implementation is valid, the CG generates a set of Java source files that implement the container of the component. These Java files (see Figure 7.2) are compiled together with the implementation code to generate a standalone and ready-to-use component. Thanks to the component container that embodies all resources needed to adapt the implementation code to the run-time environment. In particular, the generated container embodies:

- mechanisms to instantiate, connect and run the component;
- a local facet for the business code developer who does not need to have a consistent knowledge about the underlying infrastructure;
- a server facet that is connected to the local facet with a facade;
- a facade that transforms the output invocations in the local facet to an output service call emitted by the server facet (and vice versa for the inputs);
- scheduling mechanisms to control the execution of the service invocation queue.

Figure 7.3 shows the process that is followed to generate the container. It also shows how the DDF rules are used and the way in which the process steps are performed (i.e., automatically or manually).

During the execution, a particular component runs by default. This component, called Component Manager (CM), supports the creation of components and establishes connections between them. To make the connection between two components, the CM uses their two interface description files to match the required and provided services for both components. This matching works both ways.

They exist two configurations in SON middleware. The first configuration (local) can manage the local exchange between the components on the same peer. In this case, the CM manages locally a list of components. The second configuration allows managing the publishing and discovery of components in a P2P network. In this context, the CM delegates the management of remote component lists to a DHT (Distributed Hash Table) [Rhea et al., 2004]. A DHT is a distributed system that provides mechanisms to collectively manage a mapping from hash values (keys) to some kind of content (data values), without any centralized control or fixed hierarchy, and with a little human assistance. DHTs were introduced in the research community of P2P because, in most cases, the challenges of P2P systems (e.g., storage, connectivity, coordination of resources, etc.) can be reduced to a single problem: *"How do you find any given data item in a large P2P system in a scalable manner, without any centralized servers or hierarchy?"* [Balakrishnan et al., 2003].

After the connection process, two components interact with each other directly without going through the CM (cf. Section 7.3.3). The advantage of this environment is its dynamic

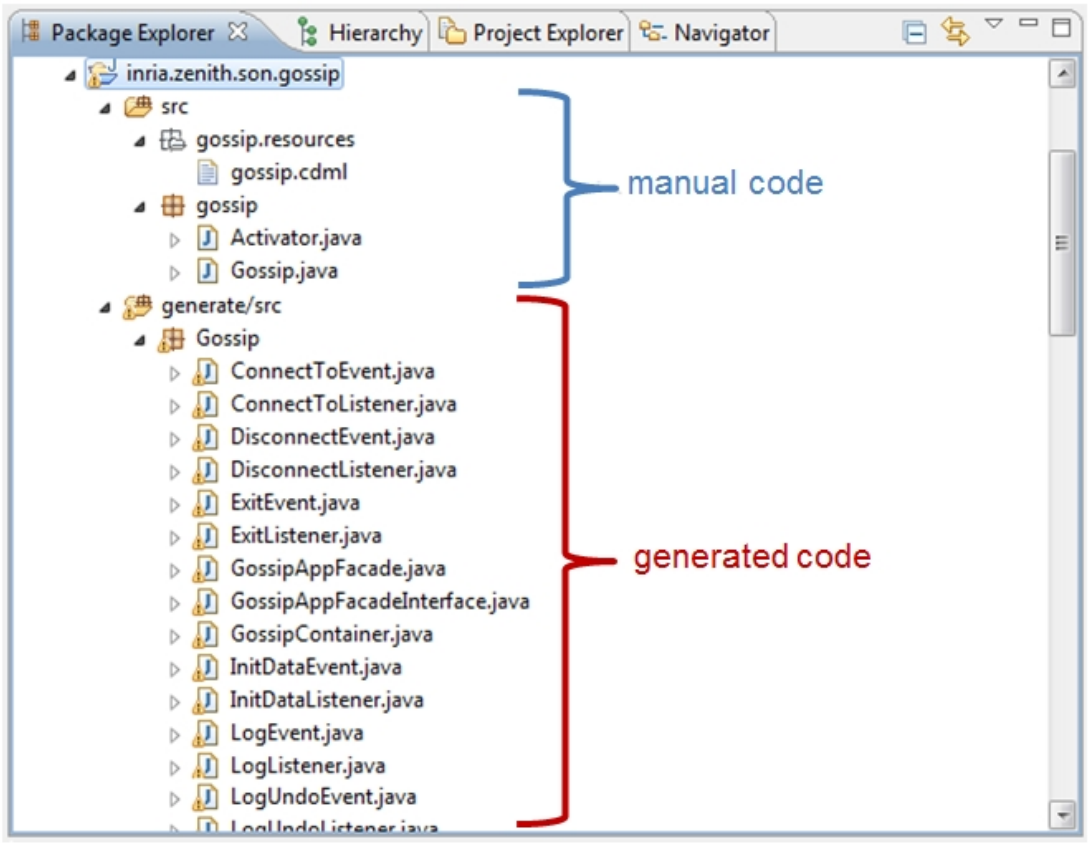

Figure 7.2: SON's component structure.

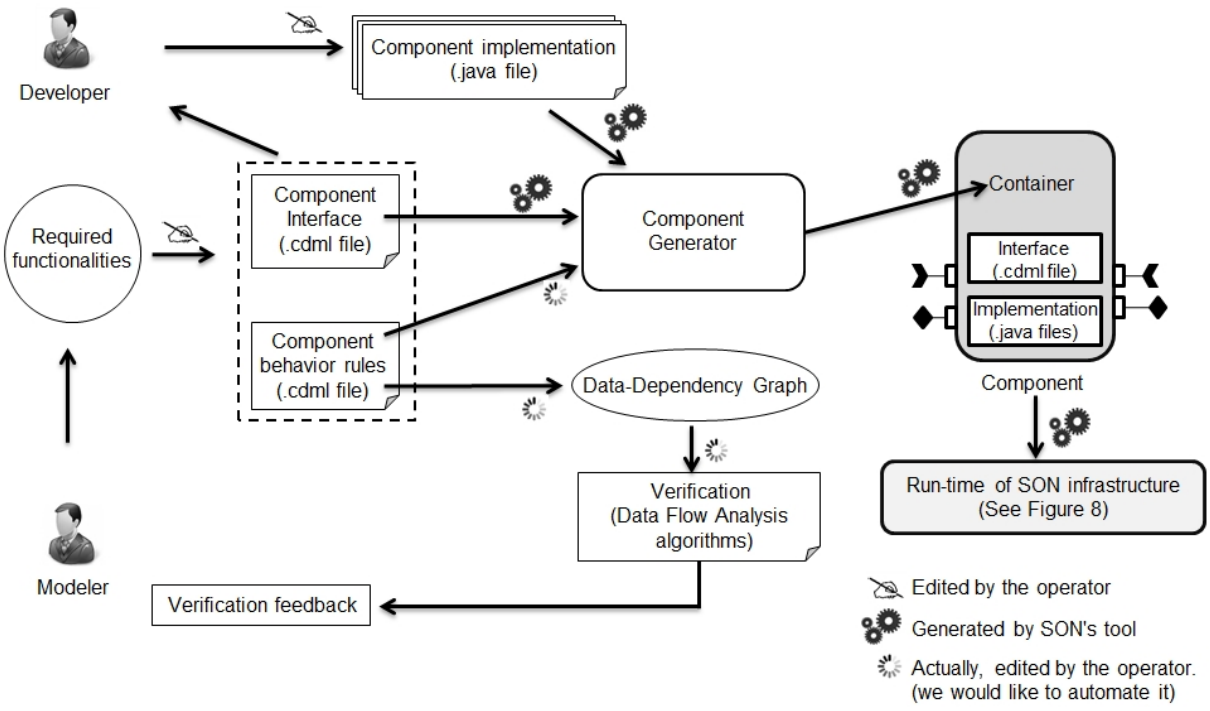

Figure 7.3: Overview of the development process.

aspect. In fact, during the execution, the components can dynamically join and leave the system over connections established on the fly. The next sections present the different aspects of this middleware in more details.

7.2 Service-oriented component model

As presented in [Liu et al., 2006], service-oriented component approach help developers to build SOC applications by separating non-functional requirements from business logic. To implement such applications, one must take into account standards, code distribution, deployment of components and reuse of business logic. To cope with these changes, applications need to be more open, adaptable and capable of evolving. We present in this section a service-oriented component model based on: i) the component interface description, named CDML and ii) the deployment description, named World.

7.2.1 The component interface description (CDML)

We have defined an abstract Component Description Meta Language, i.e., independent from any component technology:

- To enable that the runtime environment can be taken into account without any modification to the business code.
- To enable that an interface can dynamically be discovered and adapted.
- To add meta-information to a component. This is a generic approach to record information dealing with several concerns such as deployment management and component behavior (specified with DDF rules in our case).

When these mechanisms are included, The Component Generator can automatically produce the non-functional code. That is to say the container that hides all the communication and interconnection mechanisms like the transformation of a service call by a sending message, the management of a queue of received messages, and the broadcasting of a message toward the connected components. Those runtime operations are totally transparent for the designer.

As an example, a simple CDML of a component in a *Gossip system* (specified with DDF formalism in Section 5.2) is given in Figure 7.4. The *input* keyword corresponds to a provided service definition, and the *output* keyword corresponds to a required service definition. The CG can automatically generates an equivalent description in Web Services format (WSDL) when generating the non-functional code.

```
<component name="node" type="Node" extends="abstractContainer"
      <containerclass name="NodeContainer"/>
      <facadeclass name="NodeFacade" userclassname="NodeImpl"/>

      <input name="gossip" method="passiveGossip">
           <attribute name="buffer" javatype="java.lang.String"/>
      </input>

      <input name="answer" method="passiveAnswer">
           <attribute name="buffer" javatype="java.lang.String"/>
      </input>

      <output name="gossip" method="activeGossip">
           <attribute name="buffer" javatype="java.lang.String"/>
      </output >

      <output name="answer" method="activeAnswer">
           <attribute name="buffer" javatype="java.lang.String"/>
      </output >
</component>
```

Figure 7.4: Simple CDML of a component (node) in a *Gossip system*.

7.2.2 The deployment description (World)

The deployment description file is used to describe the initial state of an application. It contains a description of the components and connections that have to be created by the CM to launch the application. Of course, after that, other components can ask to be connected with each other dynamically as explained in the next subsection. A component instance is identified by the couple (name of the component, name of the instance). For example, in Figure 7.5 the instance (cmp1, cmp1-1) corresponds to an instance of component cmp1.

```
<world>
 <connectTo id_src="ComponentsManager" type_dest="cmp1" id_dest="cmp1-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1" type_dest="cmp2" id_dest="cmp2-1" />
 <connectTo type_src="cmp1" id_src="cmp1-1" type_dest="cmp2" id_dest="cmp2-2" />
</world>
```

Figure 7.5: Example of a deployment description file.

7.3 P2P communication model

7.3.1 The Components Manager (CM)

The Components Manager loads components, creates their instances and maintains a local list of them. To establish connections between two instances, the CM uses their interfaces to connect output connectors (vs. input) of the first one with input connectors (vs. output) of the second one. When connected, the two component instances interact with each other directly without going through the CM (see Figure 7.6). Connection management, which includes creation or destruction of connection, occurs when the CM receives notifications announcing changes in the component registry. These mechanisms allow an application to be built as interconnected component instances which can adapt dynamically to their context. Thanks to the CM that monitors the execution context and acts on the components by managing their connections.

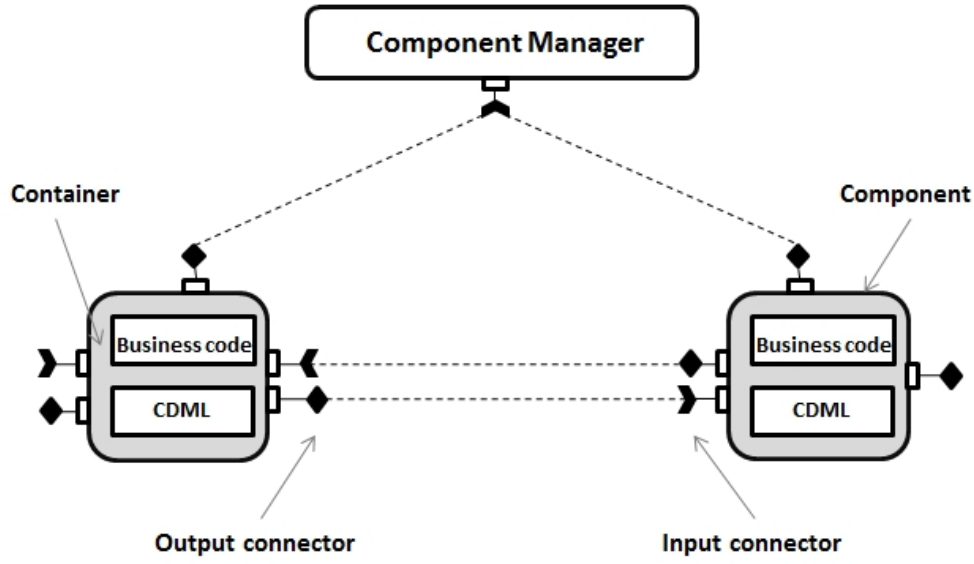

Figure 7.6: Connection between instances of components.

In P2P mode, to know whether an instance is already created, the CM should not be limited to a local search. If the instance does not exist locally then the CM should also extend the search to all connected CMs. For better modularity and information management, the CM delegates the management of components and instances tables to the DHT module. The CM has a policy to choose the effective connection. For example, a policy will favor local connections over distributed connections. Moreover, the CM structure allows to instantiate different policies by using the Command design pattern [Gamma et al., 1995]. In fact, The request to connect components is done in two steps. In the first step, the CM interrogates the local list and DHT module on the presence or not of the instance of the destination. Each one responds asynchronously to the CM. When the CM is in possession of all responses (even negative) then in the second step, it selects according to its policy the module that handles the effective connection. If in the first step, there is no positive response, the connection request is put on hold until the CM receives a notification, such as a component has been started or discovered.

To publish, discover and connect components on the network, two modules are proposed (see Figure 7.7). DHT module publishes and discovers components, and PIPES module connects components that are deployed on remote peers.

7.3.2 The DHT module

DHT module manages remote component lists. In the current version, DHT module uses the OpenChord implementation [Stoica et al., 2001], but nothing prevents from using other implementations. For this purpose, an interface was defined with the usual methods (*put (key,value)* and *get(key)*) that can be expected from a DHT module. At each creation of a component instance, the CM publishes into the DHT, the necessary information used by remote PIPES modules to establish connection to this new created component instance.

7.3.3 The PIPES module

The PIPES module handles the communication between remote component instances. It opens a TCP connection between peers. It is based on the concept of virtual pipes introduced into the JXTA [Wilson, 2002], a communication technology that has been widely used within the

Grid community. This concept allows passing through a single TCP connection, several logical communications (virtual pipes) between peers. By using this abstraction, each component may open a virtual pipe to read messages sent to it. A virtual pipe is identified by a Universally Unique Identifier (UUID). This identifier is associated with the component instance name and registered in the DHT as follows:

[Key: component instance Name, Value: UUID of the virtual pipe]

[Key: UUID of the virtual pipe, Value: UUID of the PIPES module]

[Key: UUID of the PIPES module, Value: IP + Port Number]

The second record associates the virtual pipe component with the PIPES module it belongs. The third record associates the PIPES module with its IP address and port number. Thus, two peers can find into the DHT all the information needed to connect their components.

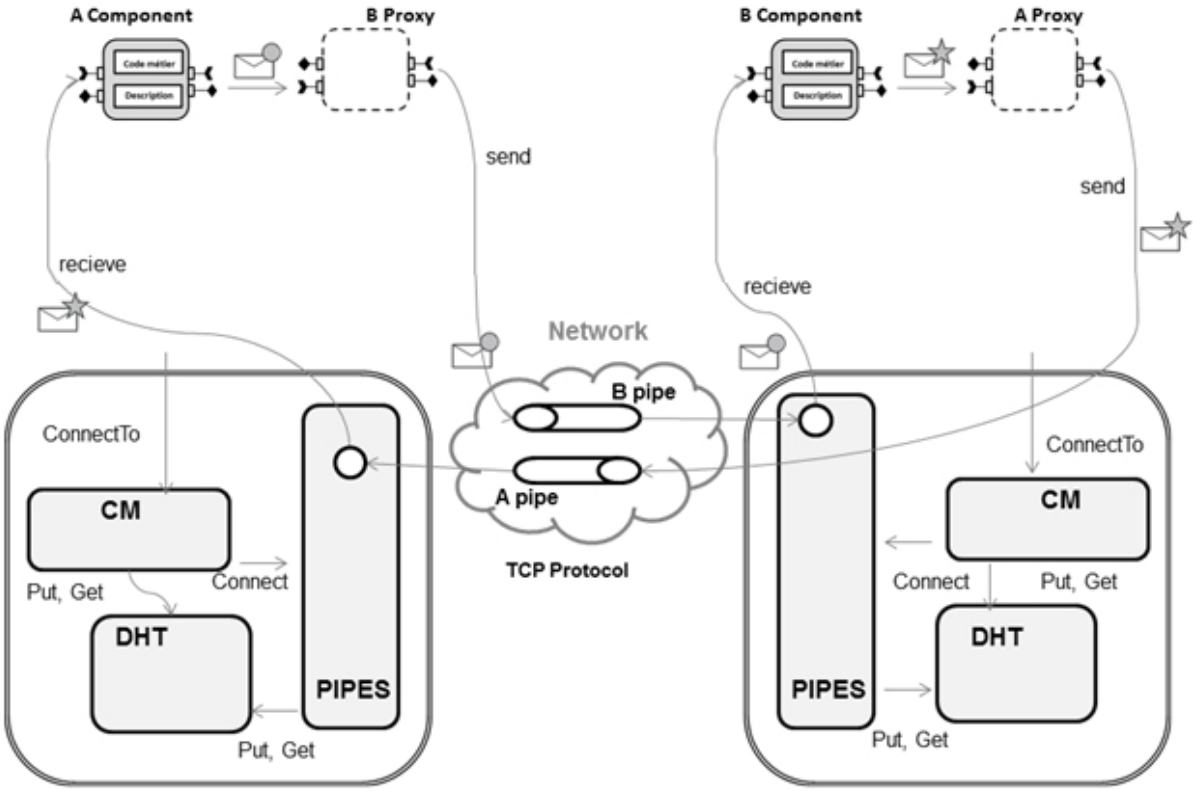

Figure 7.7: Run-time architecture of SON middleware.

7.4 Implementation

This approach has been fully integrated into the Eclipse environment [The Eclipse Foundation, 2003] and implemented on top of OSGi [The OSGi Alliance, 2007]. Eclipse is built around a very small extensible runtime core and its functionality, (including compilers, workbench, and support tools) consists of plug-ins that can be managed separately. That allowed us to integrate the Component Generator (CG) into Eclipse as a plug-in.

The application programmer develops his Java code with Eclipse IDE, in the classic way. Then, after defining the CDMLs, non-functional codes are generated using the CG plug-in to obtain components usable by the SON middleware (see Figure 7.2). The OSGi service platform provides a computing environment for applications, called bundles, to dynamically deploy services in a centralized environment. It is also a small layer that allows multiple components to efficiently cooperate in a single Java Virtual Machine (JVM) by managing aspects of local service deployment. However, OSGi service platform leaves service dependency management as a task for component developers, thing which is treated automatically in our case by the CM.

At the start of execution, the OSGi platform is launched, and the CM is started by default as a bundle. In this context, two OSGi services are used and published. The first one, called *ContainerService*, allows publishing the CDML when a component is started. The CM then adds that started component to its table of available components. The second one, called, *ContainerProxy*, allows publishing the component instance when it is created. The CM then adds that new instance to its table of created instances. The CM can then manage the execution in an extended environment unlike other classic Java application environments. Moreover, installing a new bundle, registering a new service, or updating an existing component does not need a restart of the JVM because the concerned components are notified of the new state and adapt their connections accordingly through the CM.

7.5 Applications

In this section, we illustrate the practical use of SON middleware with two application scenarios: i) Simple Georeferencing Tool, ii) P2Prec: a social based P2P recommendation system for

large-scale data sharing. For each of these applications, we briefly describe its principle and how SON has been used.

7.5.1 Simple Georeferencing Tool (SGT)

Simple Georeferencing Tool (SGT) is a lightweight prototype implemented as an application of SON middleware. It is only composed of three SON's components. SGT is dedicated to collect, process and display georeferenced individual level data. Georeferencing is relating information to geographic location [Hill, 2006] and its scope includes the informal means of referring to locations, which we use in ordinary discourse using placenames, and the formal representations based on longitude and latitude coordinates and other spatial referencing systems.

The application of georeferencing extends to almost all fields of human activity, including medicine, agriculture, petroleum exploration, government administration and historical research.

Georeferencing tools include services to identify a location of a place, object or person, such as discovering the nearest gas station or the whereabouts of a colleague or friend. They include package and vehicle tracking services, location-based games and even marketing services. In our case, we have chosen to explain our simple georeferencing tool SGT by using it as a geo-recommendation application as described in the following scenario.

A simple application scenario: using SGT for Geo-recommendation

In cities all over the world, people search to discover new places, to describe their impressions and to share their discoveries with their colleagues, family, and friends. SGT is used to create and display a combined view of surrounding addresses along with recommendations based on the experiences and tastes of other persons. Thus, when SGT users are far from home and need information about new places (restaurants, movie theaters, museums, gyms, etc.), SGT's search engine can help them with the recommendations of locals in the surrounding area.

In this experimental scenario, SGT implementation consists in three SON's components: *Provider*, *Consumer* and *Super-node*. *Provider* component instances are used to expose the

georeferenced services to the network, while *Consumer* component instances are used by service consumers. Each *Super-node* instance is responsible for serving a certain number of *Provider* and *Consumer* instances by publishing georeferenced services and the associated recommendation, answering queries, and creating notifications. *Super-node* component embodies the functionalities of SON's communication model (see Section 7.3). Thus, and instead of using a central server as the case of most georeferencing tools, *Super-node* instances form an overlay network based on a DHT that offers a reliable, robust and scalable mechanism to store and manage data using P2P principles. As indicated in Section 7.3.2, we use OpenChord as a DHT implementation.

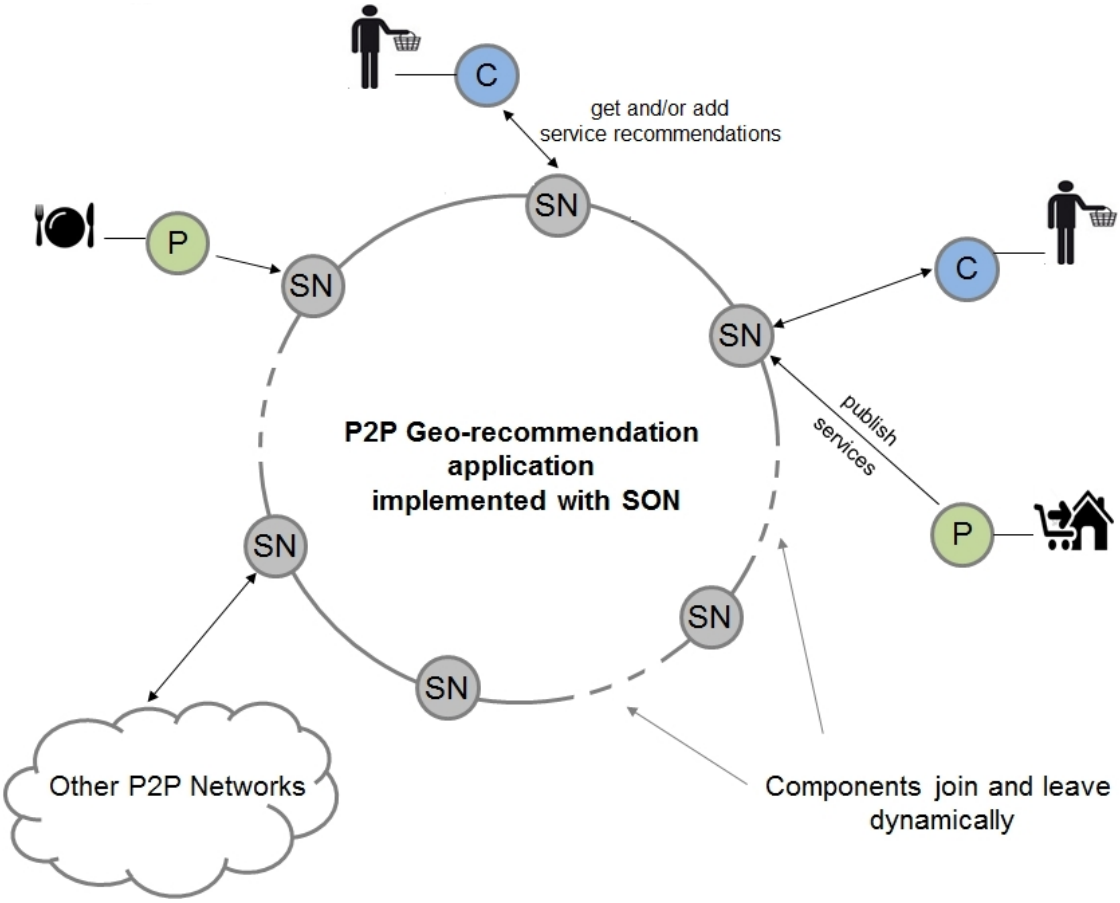

Figure 7.8: Using SON to implement a geo-recommendation application.

Figure 7.8 gives a simple use case where provider users (a restaurant and a shop) use *Provider* component instance (denoted by *p*) to publish their georeferenced services, while consumer users use *Consumer* component instance (denoted by *c*) to get and add recommendations about those services. *Provider* and *Consumer* instances connect to the network through

Super-node instances which are their access points.

Provider users are required firstly to add (through *Provider GUI*, see Figure 7.9) new places on the map and submit some information about the services available in those places. Places on the map can be a local, work zone, district, path, department, etc. The service information contains a name and a brief description.

After that, for each georeferenced service, a *key-value* pair is stored in the DHT. The *key* is calculated depending on the longitude and latitude coordinates of the region where the service place is located. The *value* of a *key* has the following form: *serviceInfo, point, point, point, ... point*, where each *point* corresponds to the longitude and latitude coordinates of the corners of the polygon representing the service place.

Thus, a consumer user can discover the available services around him by running queries in the DHT through the *Consumer GUI* (see Figure 7.10). Afterwards, the consumer user can

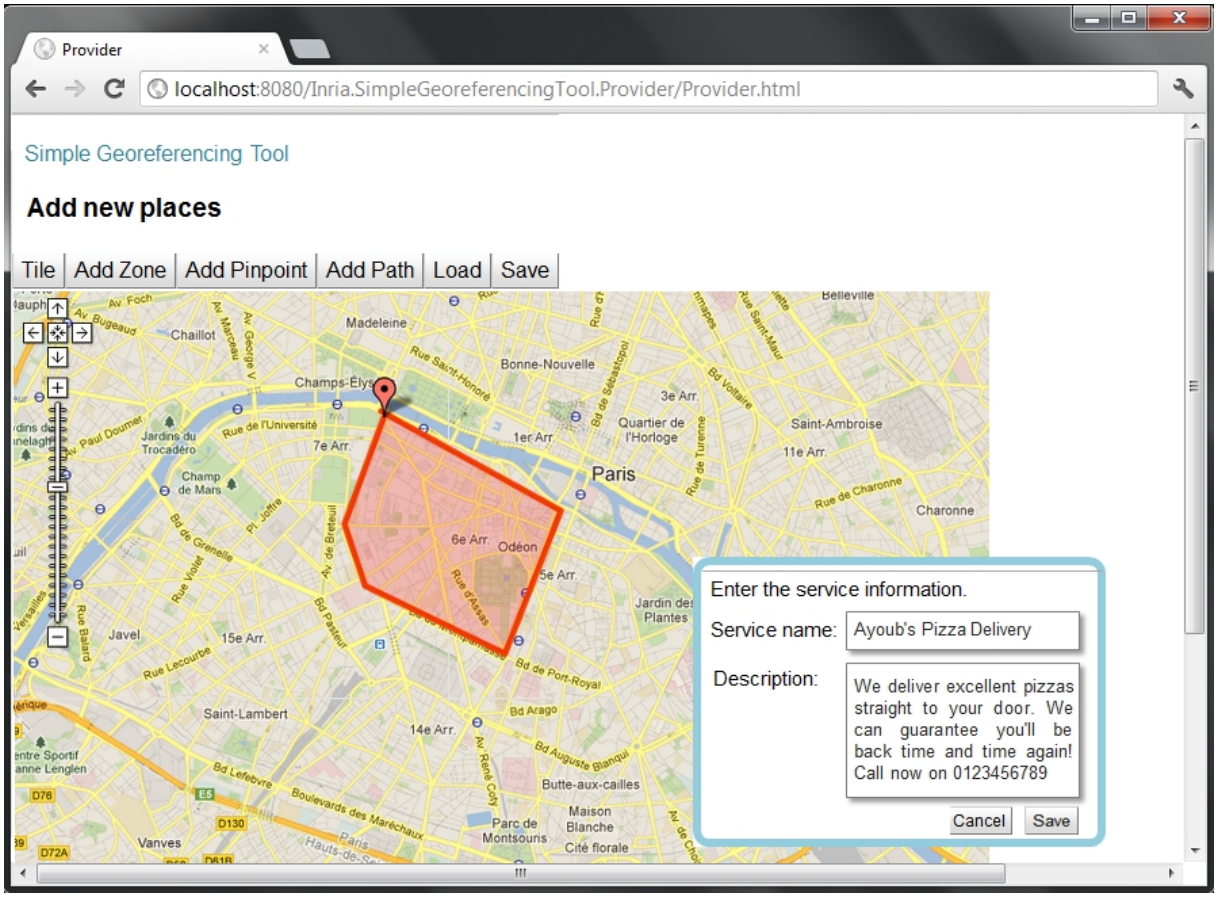

Figure 7.9: Screenshot of *Provider GUI* .

add his own recommendations about a service. He can also subscribe to a desired service and receive notifications about new recommendations added by other people.

We close by pointing out that the front-end part (*Provider* and *Consumer GUI*) has been generated from a Java Servlet using Google Web Toolkit (GWT) [GWT, 2007]. Java Servlet is a server-side web technology that serves user requests and receives responses from the business code of the component. GWT is a development toolkit for building complex browser-based applications without the developer having to be an expert in browser technologies (e.g., JavaScript, AJAX and XMLHttpRequest). GWT cross-compiler translates the Java source code to standalone JavaScript files that are deeply optimized. These allow SON's components to easily provide a web user interface that runs across all browsers, including those for mobiles.

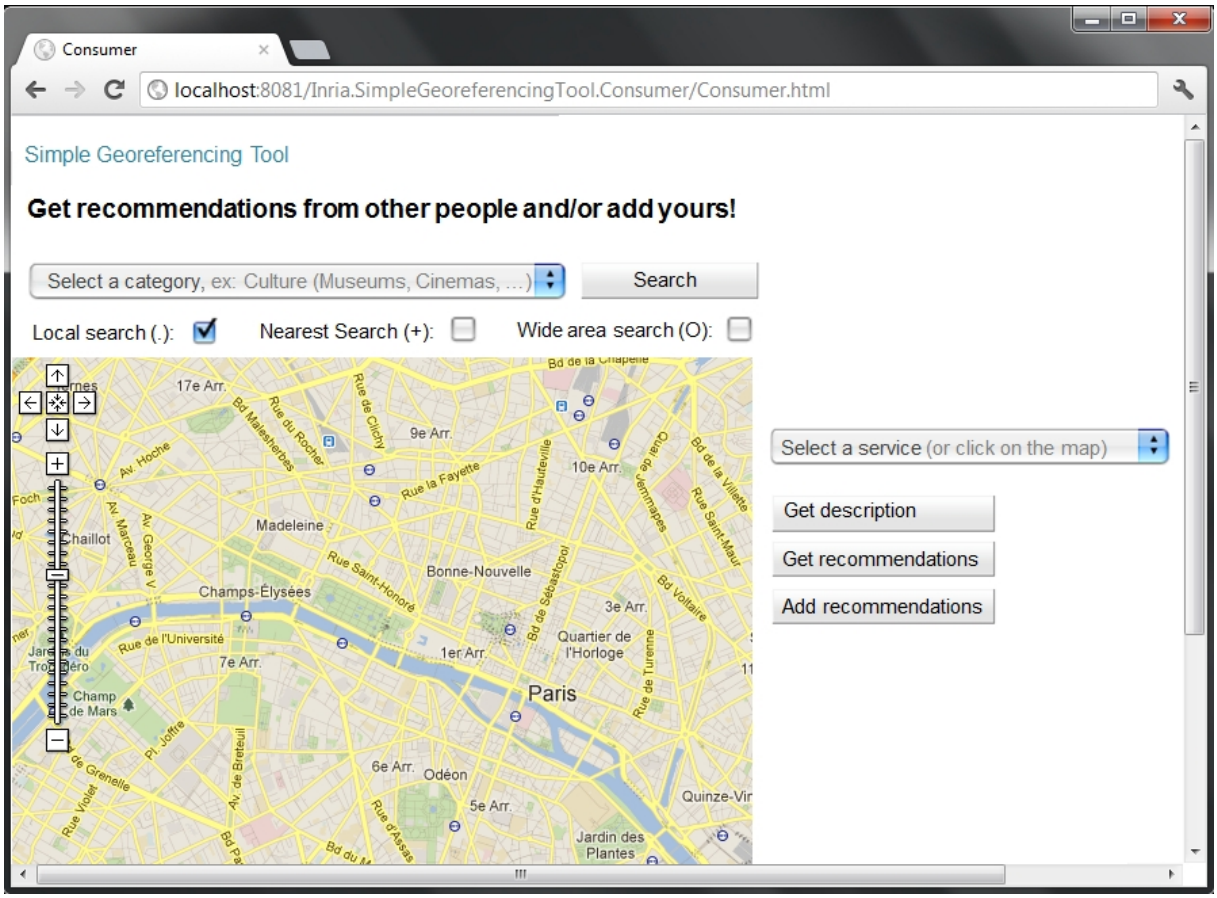

Figure 7.10: Screenshot of *Consumer GUI*.

7.5.2 Social-based P2P recommendation system (P2Prec)

In this section, we present a summary of the implementation of a prototype based on *Gossip protocol* (specified with DDF formalism in Section 5.2). The prototype is called P2Prec: a social-based P2P recommendation system. P2Prec was conceived in our research team [Draidi et al., 2011b, Draidi et al., 2011a] and its web site is available online [P2Prec, 2011]. We refer to those references for further reading.

Locating contents based on contents ids in a P2P overlay network is now well solved. However, the problem with current P2P content-sharing systems is that the users themselves, i.e., their interest or expertise in specific topics, or their rankings of documents they have read, are simply ignored. Consider, for instance, a scientific community (e.g., in bio-informatics, physics or environmental science) where community members are willing to share large amounts of documents (including images, experimental data, etc).

P2Prec is a social-based P2P recommendation system for large-scale content sharing. The main idea is to recommend high quality documents related to query topics and contents hold by useful friends (of friends) of the users, by exploiting friendship networks.

The recommendation model relies on a distributed graph, where each node represents a user (peer) labeled with the contents it stores and its topics of interests. The topics each peer is interested in are automatically calculated by analyzing the documents the peer holds. Peers become relevant for a topic if they hold a certain number of highly rated documents on this topic. A peer v becomes useful to a peer u, if u's topics of interest and v's relevant topics are overlapped. To disseminate information about relevant peers, P2Prec rely on *Gossip protocol* as follows. At each gossip exchange, each user u checks its gossip local-view to enquire whether there is any relevant user v that is useful to u, and its friendship networks have high overlap with u's friendship network. If it is the case, a demand of friendship is launched among u and v. Whenever a user submits a key-word query, this query is redirected to the top-k most adequate friends by taking into account similarities, relevance, usefulness and trust.

We developed P2Prec as a SON application with two components: the *LDA* component for the documents topics process and the *P2Prec* component for the recommendation process. For

instance, the services of the *P2Prec* component are the services for passive and active propagation through gossip services (*gossip* and *gossipAnswer* services) and the queries services (*query* and *queryAnswer* services). There are two OSGi configurations, the Bootstrap Server (BS) configuration and the Client (the peer) configuration. To run the P2Prec application, the BS must be started on a given machine (with a given IP address). This IP address will be used as the entry point into the P2Prec network for new peers. At the startup time, a new peer must first identify itself with the BS (*connect* service) and the BS is going to return the current set of all topics (*allTopics* service). Then within the local peer's *LDA* component and the current topics, the topics of each document is computed locally.

After these steps, the peer can start the recommendation steps and documents discovery without any connection with the BS. Indeed, the research of topics of a new document (*computeTopic*(*doc*) service) and the computing of topics of a query (*computeTopic*(*query*) service) can be made locally with the local peer's *LDA* component. Depending on the evolution of documents on the P2Prec network, the BS may update the set of topics of documents, and inform the peers by broadcasting this new topic set (using the *allTopics* service).

Chapter 8

Evaluation of SON in the STAMP project

Contents

This chapter aims at presenting the results of the evaluation of SON middleware in the context of the STAMP project. The evaluation consists of implementing application scenarios from the area of modelling environmental landscapes and their dynamics. The objective is to show how SON middleware (especially the dynamic availability of services in a service-oriented runtime) is able to improve and enhance the effectiveness of such environmental application scenarios. At first, the main characteristics of today's research approaches in environmental modelling are outlined in Section 8.1. Known limitations of these approaches have led to the initiation of the STAMP project, whose objectives are summarized in Section 8.2. The

fundamental concepts of a modelling language (called Ocelet) developed to meet the STAMP objectives are outlined in Section 8.3. Finally, two environmental application scenarios are presented in Section 8.4: i) the Lotka-Volterra model which is also called predator-prey model, ii) a land-scape modelling experiment that consists on the spread of a mosquito-borne disease (Rift Valley Fever) in an arid area, in West Africa, where ponds, pastures, herds and mosquitoes come into play.

8.1 Characteristics of the main research approaches in environmental modelling

Computer modelling of systems in space and time is common practice in many scientific disciplines. It allows by simulation the verification of the knowledge one has of a system, and therefore helps to better understand how the system works in some situations, while aiming at predicting the behavior of the system in a variety of other situations. When the system considered is an environmental landscape, for which full scale physical experimentation can rarely be considered, modelling could be applied to help analyze a variety of important issues facing society today, such as the degradation of natural ecosystems with loss of biodiversity, the emergence and spread of new diseases due to changing environmental and climatic conditions, or the uncontrolled urbanization and population migrations as expressions of deep social transformations.

The modelling of spatial and non-spatial dynamics of landscapes have been carried out in a large variety of not only thematic, but also methodological contexts. No less than five paradigms or modelling formalisms – system dynamics (SD), discrete event (DE), cellular automata (CA), agent-based (AB) and geographic information systems (GIS) – are being used [Burrough and Mcdonnell, 1998, Borshchev and Filippov, 2004, Bousquet and Le Page, 2004, Ratze et al., 2007]. This diversity, while being a sign of an active research field, may also suggest that the concepts used by modellers could be too diverse to be satisfactorily described with any single formalism. For instance, while geared for manipulating spatial information, GIS suffer from an intrinsic limitation of not properly handling time (e.g., [Langran, 1992]). During the last two decades there have been major contributions to address the Time issue

in GIS (e.g., [Langran, 1992, Peuquet, 1994, Worboys, 1994, Claramunt and Thériault, 1995, Yuan, 1999, Wachowicz and Wachowiez, 1999, Parent et al., 2006]). Adding Time as another dimension to space proved however not to be just an implementation problem, and recommendations were made that more theoretical and conceptual developments would be required [Peuquet, 2001]. Likewise, formalisms that consider Time first (i.e., SD, DE) face the opposite limitation with spatial information, where it is widely assumed the latter can only be treated as either field or object models [Goodchild, 1992, Peuquet, 2001]. Improvements were sought with coupled or hybrid models that capitalize on more than one of the formalisms: SD-DE (e.g., [Zeigler, 1984]); AB-SD (e.g., [Duboz et al., 2003]); GIS-AB (e.g., [Brown et al., 2005, Torrens and Benenson, 2005]); AB-DE (e.g., [Uhrmacher and Schattenberg, 1998]); AB-CA (e.g., [Bousquet et al., 1998]); CA-DE (e.g., [Wainer and Giambiasi, 2005]). These works are representative of what can be considered a highly active research domain, where research communities assemble to address common thematic (e.g. landscape ecology, urban planning and management, spatial epidemiology), methodological (e.g. MAS–multi-agent systems, DEVS–discrete event system specification) as well as conceptual (e.g. object-field models of space (e.g., [Couclelis, 1992, Cova and Goodchild, 2002]), hierarchy and scales (e.g., [Wu, 1999]), data quality (e.g., [Devillers and Jeansoulin, 2006]), indeterminate boundaries (e.g., [Burrough et al., 1996], time in GIS) issues.

In addition to the problem of choosing the appropriate modelling approach in a given context, previous studies have stressed on the difficulties that modellers face when working from conceptual models of dynamic landscapes to their simulation on a computer [Fall and Fall, 2001]. A general-purpose modelling language such as UML [OMG, 2007] appears unsuitable for two main reasons: (i) it is not a directly executable specification: the execution model is only partially implemented, such that the user must manually complete the produced code and (ii) the concepts proposed are very general, and not readily configurable to the present case. One approach has been to develop domain specific languages (e.g. SME; [Maxwell and Costanza, 1997]) (SELES; [Fall and Fall, 2001]) (L1; [Gaucherel et al., 2006]) that would allow domain experts to concentrate on the conceptual model, while leaving to an associated software tool the transformation of the model into an implementation that runs on a computer. In this way, domain experts may develop models using a higher level language, instead of programming directly

with general-purpose languages like Java or C++. However, for such a large domain where spatial, temporal and multi-scale issues are still actively being studied, a DSL that can support research on modelling processes in landscapes has to be flexible, and especially so at the very basic level where landscape features and their interactions are defined. For example, a DSL that has originally been developed using a predefined spatial data structure (e.g. grid cells) may limit modellers in situations where other structures are more appropriate [Gaucherel et al., 2006]. A trade-off between ease of use and expressiveness of a DSL therefore seems inevitable here.

An interesting parallel can be made between, on one hand, landscape entities and their interactions that need to be modelled, and on the other, software components and services that emerged with the component model programming. Interacting features in a landscape in many aspects behave like communicating software components, and it is not surprising that many notions used when modeling processes occurring in landscapes, such as dynamics, delays, events, response or agent behavior, are also present in the service-oriented computing (cf. Section 3.2). In this study, we present an approach which has been developed for experimenting the modeling of a variety of landscape situations, while taking advantage of the power of component-service programming.

8.2 The STAMP project

8.2.1 Factual information on the project

The STAMP project (STAMP: modelling dynamic landscapes with **S**patial, **T**emporal And **M**ulti-scale **P**rimitives) is a research project coordinated by Danny Lo Seen (from CIRAD, a French research centre working with developing countries to tackle international agricultural and development issues). STAMP was supported (in part) by the Agence Nationale de la Recherche (ANR) under Project No. ANR-07-BLAN-0121. The project partners are the TETIS research unit (CIRAD; team leader: Pascal Degenne), the Zenith team of INRIA (team leader: Didier Parigot), the Gaspard Monge Computer laboratory of Paris-Est University (team leader: Olivier Curé) and the AMAP research unit (INRA, IRD; team leader: Daniel Auclair).

8.2.2 Goals of the project

Each of the approaches discussed above in the previous section has demonstrated specific benefits in different domains of application. However, research on environmental modelling remains organized around tools that are not quite inter-compatible, in very dynamic but separate research communities, whereas integration of different disciplines is crucial given the important challenges facing societies today.

A known limitation of these approaches is the strong constraint relative to the format used to represent spatial entities, urging the modeller to think in terms of grids, points, lines or polygons. STAMP project attempt to overcome this constraint by exploring an approach based on the use of modelling primitives. In the process, it was necessary to identify and define concepts that are essential for modellers, then build a modelling computer language (called Ocelet), together with the grammar and syntax needed to manipulate these concepts, and finally, to develop the compiler and the environment/interface for building models and running simulations. With Ocelet, the landscape is seen as a system composed of entities that interact through relations. The language allows using pre-developed primitives to describe these entities, the relations that link them, and to establish evolution scenarios of the system.

8.2.3 Our contributions in the project

Within the STAMP project, we have contributed in two main ways. First, we have participated in the design and the specification of the Ocelet modelling language. Second, we have defined for Ocelet a service-oriented component runtime, based on SON infrastructure.

Compared to similar existing languages (cf. Section 8.1), Ocelet language elements have been designed to seek a balance between modelling facility in simpler situations and adequate expressiveness in more complex ones, while taking advantage of the power of component-service programming. The structure and the logic of the language, as well as the language elements, are introduced in Section 8.3. For further reading, we refer the reader to [Degenne et al., 2009] and [Degenne et al., 2010].

In addition to landscape complexity that is difficult to address otherwise than by modelling,

landscape scientists (from different disciplines) face other difficulties when working from conceptual models to their simulation and execution on a computer. Moreover, such highly dynamic applications must be able to adapt according to their own evolution during the execution. Beside, various businesses (the domain of landscape modelling) and technical challenges (the management of dynamism and service interactions) complicate the ability to develop this kind of applications. For these reasons, we have chosen to provide an Ocelet runtime that would allow domain experts to concentrate on the conceptual model, while leaving to SON middleware the transformation of the model into an implementation, which runs on a dynamic execution environment. Thus, the modeler of an Ocelet application does not need to know how the non-functional code (e.g., communication mechanisms, sending and receiving messages, message queue management, etc.) is implemented. Furthermore, the encapsulation of Ocelet elements in SON components allows the reuse of these elements in other landscape models. Since SON has been presented in the previous chapter, we only highlight in Section 8.4 application scenarios from the area of landscape modelling in order to demonstrate the capabilities of the Ocelet runtime based on SON middleware. For further reading, we refer the reader to [Ait Lahcen et al., 2009] and [SON, 2011].

8.3 Ocelet modelling language

Ocelet has followed DSL development procedures recommended by [Mernik et al., 2005]. Its design had to meet two main requirements: i) it has to provide concepts adapted for modelling processes in landscapes, and ii) it must have underlying operational semantics that are able to automatically generate code and run simulations corresponding to the models written with the language. Around the language there is a modelling framework composed of

- a model building environment that enables syntax analysing and type verification,

- a code generator and compiler, and

- a program execution runtime based on SON infrastructure (see Figure 8.1).

Ocelet is designed around five main concepts: *Entity, Service, Relation, Scenario* and *Datafacer*. We define hereafter how these concepts should be understood in the context of

Ocelet. Other common concepts such as argument, property, number are also used, but they do not require specific descriptions.

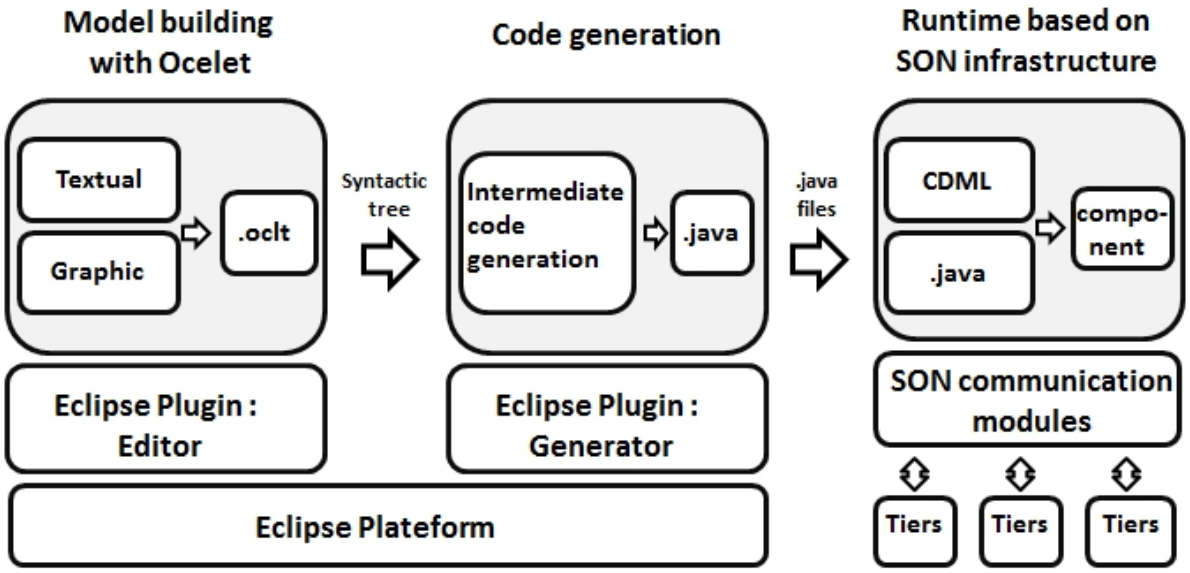

Figure 8.1: The *Ocelet* modelling and simulation framework.

8.3.1 Ocelet main concepts

Entity

Entities are basic modelling parts that can be put together to build a model. A whole model is, as such, also an entity. An entity can contain other entities, and is then called a *composite* entity. Entities that do not contain other entities are called *atomic* entities. A forest for example can be modelled by a composite entity that contains tree entities which are part of the forest.

From a computer science point of view, an entity is a component: an independant piece of code that can be connected to other components to build an application. Entities can perform operations called services. Entities being software components, they can dynamically be connected through their services, even without knowing how they are designed internally.

Structure of an entity:

```
entity( name, property*, service*, entity*, scenario*, relation*,
        datafacer* )
```

That specification means that an entity can contain properties (`property*` means 0 or more property), services, entities, scenarios, relations, datafacers, and a name. Figure 8.2 gives an illustration of this concept.

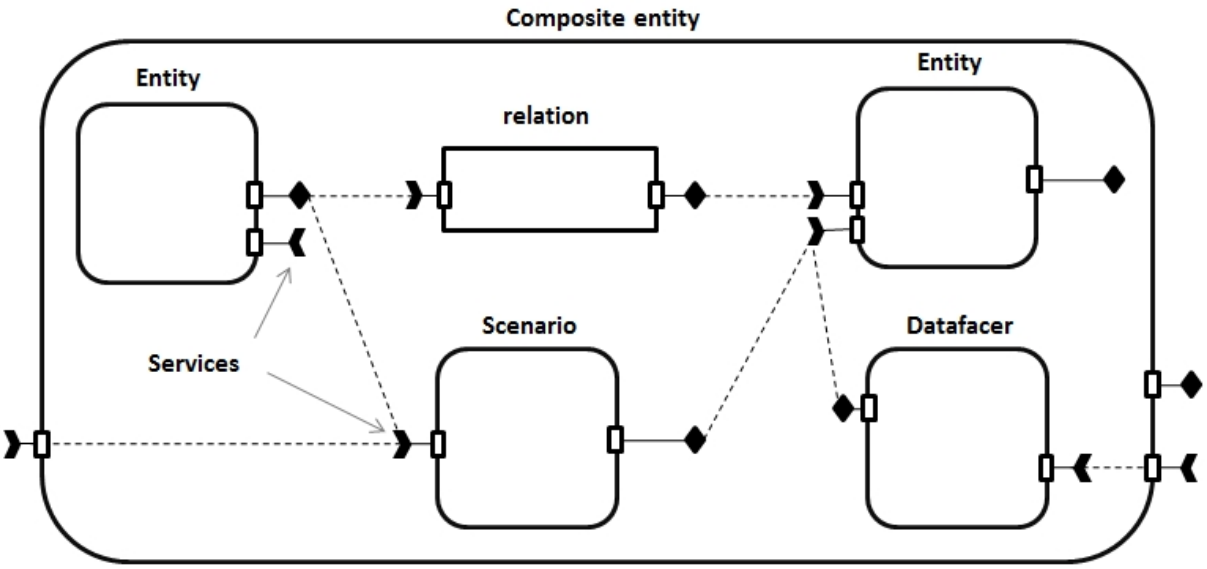

Figure 8.2: An illustration of a *composite* entity.

Service

A service is a functional description of how one can relate to an entity. It is thus a communication port of an entity. As arguments, service accept values from other entities, and describes the capability of the entity to export values to other entities of the model. Services are published outside the entity they belong to, meaning that it is possible to obtain a list of all the services an entity provides.

Structure of a service:

```
service( name, argument*, result )
```

Relation

An entity can directly call a service available on another entity. It is the simplest link that can be established betwen two entities. When modelling interacting landscape elements, relations often cannot be reduced to just a transfer of information between entities. Information sometimes

needs to be transformed according to the nature of the relation. Two aspects of the interactions have to be considered: we have to indicate which entities are interacting with each other, as expressed by an interaction graph, and at the same time describe what is happening when they interact. In Ocelet, the concept of Relation integrates both of these aspects.

Relations as interaction graphs

An interaction graph not only defines *who* are in relation (graph structure) but also *how* the elements relate (behaviour). When modelling the environment, we consider that working directly on interaction graphs can be useful for at least two reasons. First, acting at the most elementary level of the underlying data structure (a set of dynamic graphs) allows manipulating in a similar way different kinds of relations (aggregations, spatial, functional, ...). Second, the state of the model at any given time can be analysed using graph analysis algorithms to extract topological characteristics that emerge during the simulation. These may reflect some specificities of the model that would hardly be visible otherwise. Such analysis algorithms have for example been developed by [Batagelj and Mrvar, 1998], [Fuller and Sarkar, 2006] or [Saura and Torné, 2009].

Interaction graph with dynamic structure and behaviour

Entities of a model can, at a given time, relate to each other in diverse ways. For example, neighbourhood (where two entities are considered neighbours if they are close enough for a given distance function), aggregation (where some entities are considered parts of a larger composite entity), connectivity (where entities can reach each other if a communication route exist between them), influence (where one entity can influence the behaviour of another one) are in fact relations. For each relation, one can build a graph where the nodes are entities and the relations between entities are the arcs.

In many environmental modelling cases the graphs needed are actually hypergraphs (each arc may connect more than two nodes). Such hypergraphs can be built explicitly. For example, if we have several groups of entities connected to each other in the form of simple graphs,

one can establish another graph connecting those groups to each other at a broader scale. In that way, it is possible to consider the behaviour of entities within a group as well as between groups. But one can also build hypergraphs implicitly. For example, in the case of a spatial relation where an agricultural parcel is linked to each of its borders by one n-node arc, a graph is built using arcs linking more than two nodes. Such n-node arcs based graphs are de facto hypergraphs. Using n-node arcs can be a way to simplify the graph structure we have to manipulate in the model. Another aspect to take in consideration when modelling with interaction graphs is their dynamic nature. During a simulation, some entities can be added to the model, others can disappear, and individual relationships can be established or removed. This means that the interaction graphs are dynamic, with evolving numbers of nodes and arcs, and have changing graph topologies.

Attached to the graph are semantics that specify what happens between the linked entities when they do interact: the kind of information they exchange, the actions one performs on the other, the effects produced by the interaction on the entities and on the arcs involved. In many types of environmental models, attaching behaviour to an interaction graph is not straightforward. Sometimes the graph structure is implicit (e.g. cellular automata based on tessellations) and only the behaviour is specified. The programming work is then reduced but the specification of the behaviour is seriously constrained by the implicit graph structure. In other cases the graph structure is more versatile, and the arcs have to be tagged. At some other place in a program the definition of *how* entities relate is written and depends on the tags placed on the graph. A greater power of expression is obtained but the programming work is more difficult. In order to get the best of both solutions, it would be necessary to manipulate the graph structure and attach the behaviour semantics directly on that structure at a single place in the model's code.

Roles and re-usability

It is rare when an environmental model is original in all its parts. The most common situation is to have some parts of the model that are similar to other already existing models. Re-usability has been a key concern in software development and modelling tools as well. In the case of behaviours attached to relation graphs, two situations can be considered:

- Re-usability of a relation graph topology: It can be interesting to have ready made relation graph structures such as the 3-neighbours situations found in triangulated irregular network, the 4 or 8-neighbours situations found in grids, or also star and circular shaped relationships just to name a few. Based on the well known characteristics of such structures, one could imagine a modelling tool that provides optimized implementations for them to be used in different models.

- Re-usability of an attached behaviour: In that case we wish to be able to reuse the definition of *how* entities interact with each other when they do, in different modelling situations. To make the behaviour definition adaptable to a different context, the interaction should not be specified using the *entities* relating with each other but using the *role* they play. It would then be possible to attach a behaviour definition to a different relationship graph where entities are able to play similar roles. It also means that a behaviour defined once can be instantiated several times, on different graphs, even in one same model.

Finally, it can be noted that by designing a modelling tool with re-usability concerns as described above, it becomes possible to build sub-model libraries (named *primitives* in Ocelet) and make them available for a modellers community.

Modelling your point of view

At least two cases can be identified where the notion of point of view can take the form of semantics attached to a graph. First, when specialists of several different fields work on the same environmental model, they may share the same entities but need to describe interactions between these entities differently according to their own expert view. The nodes of the graph could be shared, but the arcs and the behaviour attached to those arcs would reflect their different points of views on the model. Second, it happens that different entities of a model have different points of view on their environment and would then have to interact accordingly with that environment. Here again the nodes of a graph could be shared but the arcs and the behaviour attached to those arcs could be specific to every point of view.

Relations are interaction graphs in Ocelet

The relation concept as defined in Ocelet is an interaction graph very close to what was discussed above: it contains the information of *who* is in interaction and also of *how* they interact. As relations have semantics attached to the arcs of their graph, they are constrained by the type of entities that can be linked. The definition of a relation has to specify the role played by the different entities involved, like for example:

```
relation RelationName[roleA, roleB] {...}
```

The statement above defines a relation of the most common kind: every arc of the graph links two nodes. The nodes will be entities; one entity playing role A and the other role B. Once defined, the relation must be instantiated, and which entities playing role A and role B must also be stated for that instance:

```
myInstance = RelationName[EntityA, EntityB];
```

The fact that relations are defined using roles makes them reusable in different contexts. A relation carefully designed with genericity in mind could then be used and adapted for several different models. To establish connections and actually build the graph, the predefined `connect()` and `disconnect()` services are available. For example, `myInstance.connect(lake,river)` implies that `lake` is an instance of `EntityA`, `river` is an instance of `EntityB` and an arc will be added to the relation graph between them. Ocelet allows to define relations holding hypergraphs directly by specifying more than two roles in the declaration statement, like for example: `relation RelationName[roleA, roleB, roleC, roleD] {...}`.

The *how* part is defined in the form of services that the modeller can write to precisely describe what happens when the entities interact. The services are written in the declaration of the relation, like in:

```
relation RelationName[roleA, roleB] {

    service foo() { roleA.doSmthg(); roleB.setVal(roleA.getVal()); }

}
```

The definition above implies that the entities playing `roleA` for that relation must provide the two services `doSmthg()` and `getVal()`, while the entities playing `roleB` must provide the service `setVal()`. `getVal()` and `setVal()` must also return and accept compatible types. These are verified when the relation is instantiated. One important point to note is that only one call to the `foo()` service is necessary to activate all the arcs of the relation graph.

Scenario

A scenario gives a description of which actions and relations within a composite entity have to be activated, and when. The relations in turn put selected entities in interaction in space and time. The scenario therefore expresses the spatial and temporal internal behaviour of a composite entity by managing the entities and relations it contains. For example, a ten year evolution scenario embedded in a village entity could describe the extension of the village by a few houses every year, taking in account population growth and several policy rules that govern spatial expansion. The ten-year scenario could also be composed of yearly evolution scenarios.

Structure of a Scenario:

```
scenario( name, operation* , scenario* )
```

In practice, a scenario can be used to describe how an entity evolves undisturbed for a given time period, and another scenario can contain the behavior of the same entity when a disturbance event arises.

Datafacer

A datafacer is a device through which entities access data. The data can be in the form of an external database or satellite image, but can also be internally generated, like in a logfile, during model execution. The datafacer contains the necessary functions, developed for specific types of data sources, to provide the services required by the entity to which it is attached. The other entities of the model can interact with the Datafacer in a coherent manner whithout having to deal with the details of how data access and queries are made. More formally, a datafacer is an atomic entity that can be accessed directly by any entity in a model.

8.3.2 How these concepts work together

Modellers who understand the landscape "system" they study as interacting landscape elements, should be able to express their understanding with Ocelet without much compromise. Landscape elements are modelled as entities, which in turn can contain other landscape elements (entities). Interactions between landscape elements are modelled using relations. The latter are not just "wires" for transferring information, but can also hold instructions on what to do when entities are in relation, thus expressing the "nature" of the relations.

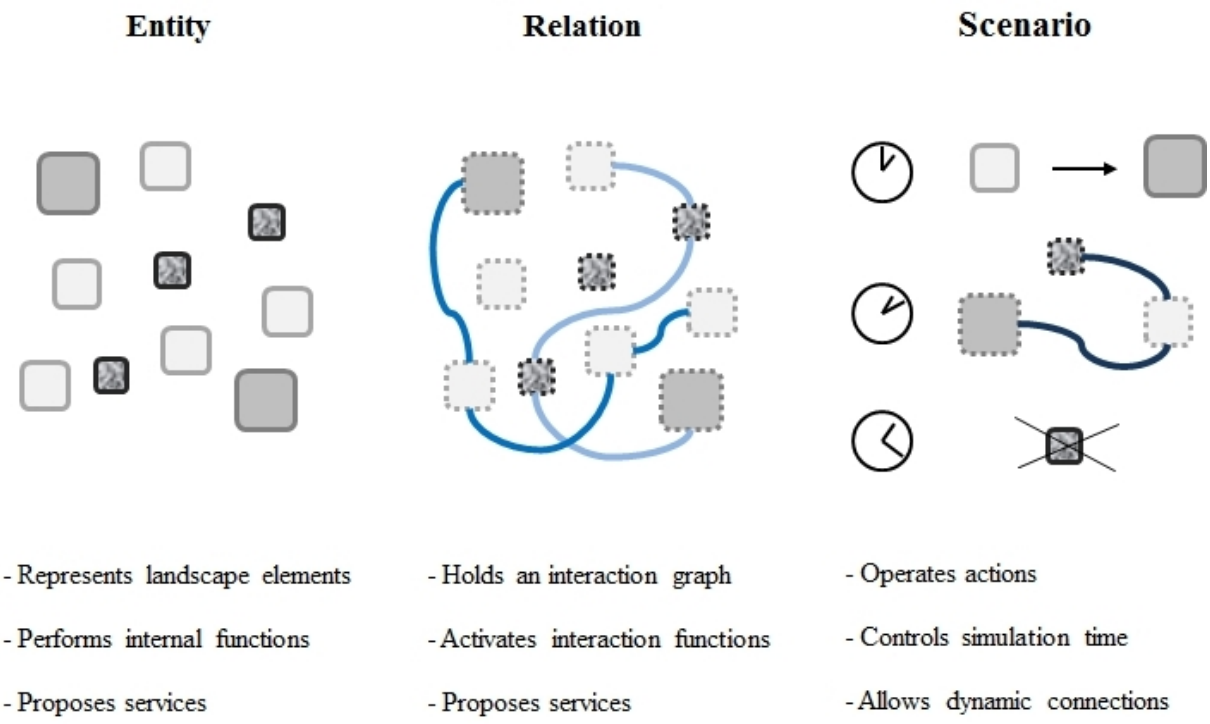

Figure 8.3: Concepts of the Ocelet language.

The orchestration of the timing of the interactions between elements in a landscape (modelled as entities contained in an entity) is carried out in a scenario attached to the landscape (see Figure 8.3). The services of an entity express the behaviour of that entity as seen from outside. Datafacers are a convenient way for entities in a model to access heterogeneous data sources through a unique mechanism based on services, and in coherence with the rest of the language.

8.4 Application scenarios with SON as a runtime

8.4.1 Lotka Volterra model

This section presents an execution scenario illustrating some requirements of landscape modeling. It consists of the well-known prey-predator model introduced by Lotka and Volterra [Lotka, 1925, Volterra, 1926, Murray, 2003] and that highlights the needs in terms of dynamicity and service interaction. The model is based on a system of non linear differential equations frequently used to describe the dynamics of ecological systems in which two species interact and evolve during time, one is a predator and one is a prey:

$$\begin{cases} \frac{dx}{dt} = x.(\alpha - \beta y) \\ \frac{dy}{dt} = y.(-\gamma + \delta x) \end{cases} \tag{8.1}$$

where

α is an expression of the birth rate in the prey population

β is the death rate of prey due to predation

γ represents the natural death rate in the population of predators

δ is the rate of predator population growth per prey consumed

Using Ocelet, two entities (*Rabbits* for preys and *Foxes* for predators) and one relation (the *Predation* relation) are defined; the time flow of the system is also described in a scenario (the *Evolve* scenario). Ocelet is designed to promote separation of concerns and in the present case the system of equations is split into the following parts:

- The birth rate of prey is calculated by the *Rabbits* entity through a *birth()* service.
- The natural death of predator is calculated by the *Foxes* entity through a *natural_death()* service.
- The death rate of prey due to predation and the growth of predator population due to predation have a meaning only if preys and predators meet in a model. They are hence

calculated in the *Predation* relation by two respective services, *updatePrey()* and *updatePredator()*.

The *Predation* relation provides a connection mechanism. When two entities are connected through it, it acts as an interposition object by providing the *updatePrey()* and *updatePredator()* services (see Figure 8.4 for the Ocelet code). This allows to enrich the connected entities without requiring changes in them. The relations therefore offer better decoupling between the business code (inside entities) and the connection code (inside relations). It is important to note that the separation between business and connection codes allows to reuse already developed relations with other entities, and in the same line, to consolidate a library of ready-made relations to facilitate future model development.

This Lotka-Volterra Ocelet model is executed above our SON middleware as follows. For each Ocelet concept: entity (*Rabbits*, *Foxes*), relation (*Predation*) and scenario (*Evolve*), that the modeler specifies using an Eclipse plugin editor developed for this need, Java files containing the translated Ocelet code and CDML files describing the services (provided and required) are generated as shown in Figure 8.1. A World file describing the initial state of the application is also generated. The component generator will then create a container for every entity,

```
relation Predation(Predator, Prey) {
    requires property number Predator.populationNbr;
    requires property number Prey.populationNbr;
    requires service Predator.updatePredPop(number);
    requires service Prey.updatePreyPop(number);

    service updatePredator(){
        Predator. updatePredPop(delta * Predator.populationNbr *
            Prey.populationNbr * dt);
    }

    service updatePrey(){
        Prey.updatePreyPop(-( beta * Predator.populationNbr *
            Prey.populationNbr * dt));
    }
}
```

Figure 8.4: Predation relation written in Ocelet.

relation and scenario. Each container encapsulates, in addition to the Java implementation and the service descriptions, all non-functional resources needed during the execution (e.g., mechanisms to instantiate, connect and run the component). Thus, we get components ready to be used or archived in the Java ARchive (JAR) files. The World file can then be used by the Components Manager to load the packages of components, create the instances, and wait a signal from the graphical user interface to start the simulation (see Figure 8.6). The interactions between predators and preys in the Lotka-Volterra model are therefore transformed into dynamic service interactions between components in a manner completely transparent to the modeler. The result of an interaction between 50 predators and 15 preys are shown in Figure 8.5 (with $\alpha = 0:1; \beta = 0:01; \gamma = 0:05; \delta = 0:001$).

Although this illustrative example may appear simple, the principal aim is to show how the Ocelet runtime allows modelers to concentrate on the conceptual model, while leaving to SON middleware the transformation of the model into a running application that take into account the requirements claimed in the context of modelling landscape dynamics.

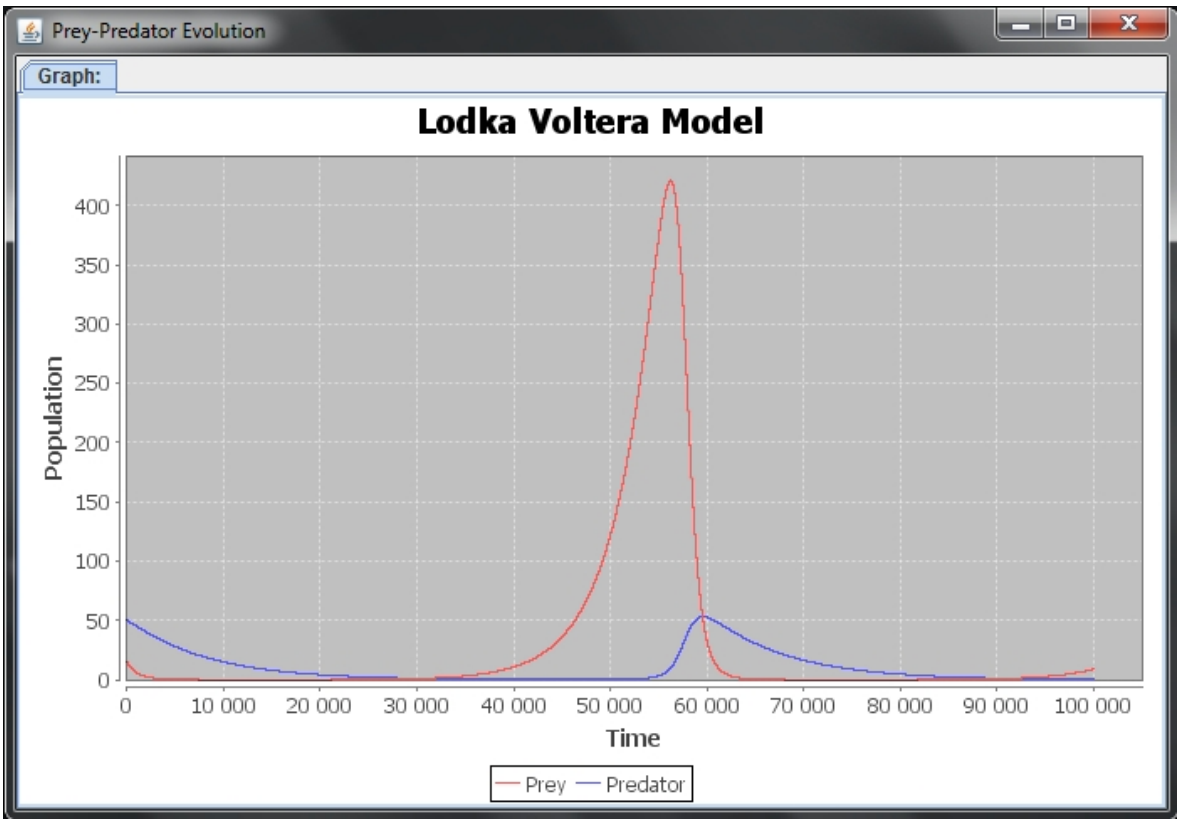

Figure 8.5: A simulation of the Lotka-Volterra model.

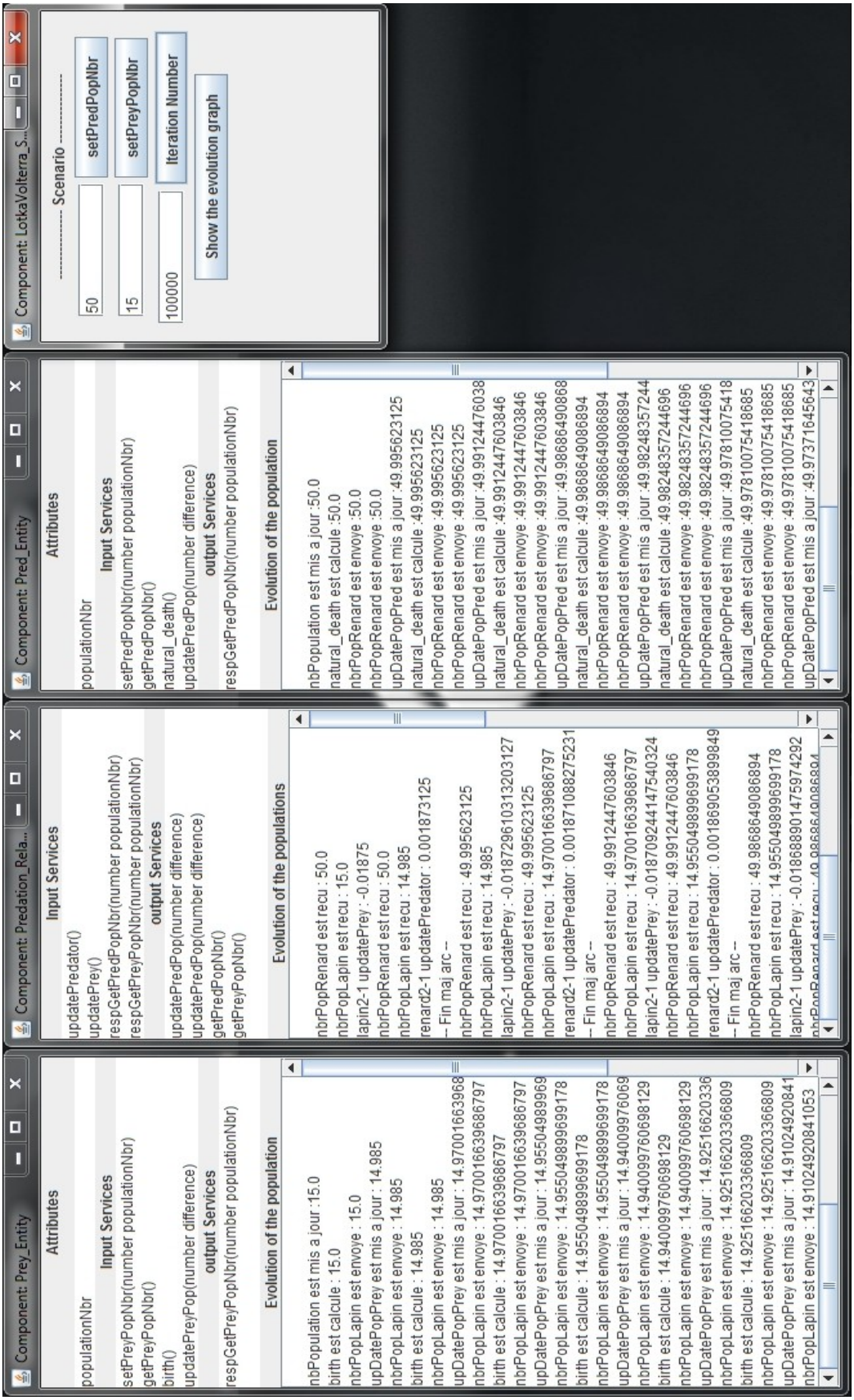

Figure 8.6: Screenshot of the GUI of the Lotka-Volterra components.

8.4.2 Rift Valley Fever (RVF), a mosquito-borne disease

The spatial and temporal distribution of mosquitoes responsible for various vector-borne diseases are often linked to landscape dynamics, as mosquitoes require appropriate breeding sites for their development. One important such disease is the Rift Valley Fever (RVF) which affects both livestock and humans. In livestock, outbreaks are generally associated with mass abortions and high mortality rates in young animals, and may result in important economic losses. The transmission of the virus in the Sahelian region of North Senegal is related to the dynamics of temporary ponds which are favorable mosquito larval habitats. The livestock production system of the region is extensive and during the rainy season, areas in the vicinity of permanent or temporary ponds are used by transhumant herds for water and grazing needs [Bah et al., 2006]. When trying to model the spread of the virus, present models, mainly epidemiological, solve Ordinary Differential Equations (ODE) for different populations of mosquito species [Gaff et al., 2007]. Most of the spatial nature of the complex problem is, however, either ignored, or concealed in appropriate contact rate parameters that are difficult to estimate. As far as we know, only few studies focused on the spatial dynamics of vectors and the disease they may transmit [Tran and Raffy, 2006, Otero et al., 2008, Linard et al., 2009]. In order to understand the dynamics of the disease in view of proposing control measures, any important aspect of the problem must not be ignored: it would be necessary to model mosquito populations according to pond dynamics and presence of livestock, and therefore also model ponds, pastures, herds that move following availability of water and food, and the transmission of the virus to the animals. The approach that we are exploring offers interesting possibilities for modelling and running complex problem simulations by focusing on each part one by one, without ignoring the interactions between the parts. In the next, we focus on some of these possibilities.

Modelling and running simple pond dynamics

In and around a given pond, the presence and abundance of *Aedes* and *Culex* mosquitoes at different life stages depend for a large part on the sequence and duration of wet and dry periods for that pond. Here we start with a simplified model of pond dynamics that describes the evolu-

tion of water surface, given the pond's shape and the quantity of water incoming or leaving the pond. The positive and negative terms of a pond's water budget are assumed to be only rainfall and evaporation respectively. Other terms such as infiltration, run-off or water consumed by animals have been ignored in this example, but could be included in a similar way. Therefore to start with, the model is made of two atomic entities: Pond and Meteo. The functioning of the model will rely on the relations between these entities, and on the scenario that describes how these relations are expressed in time.

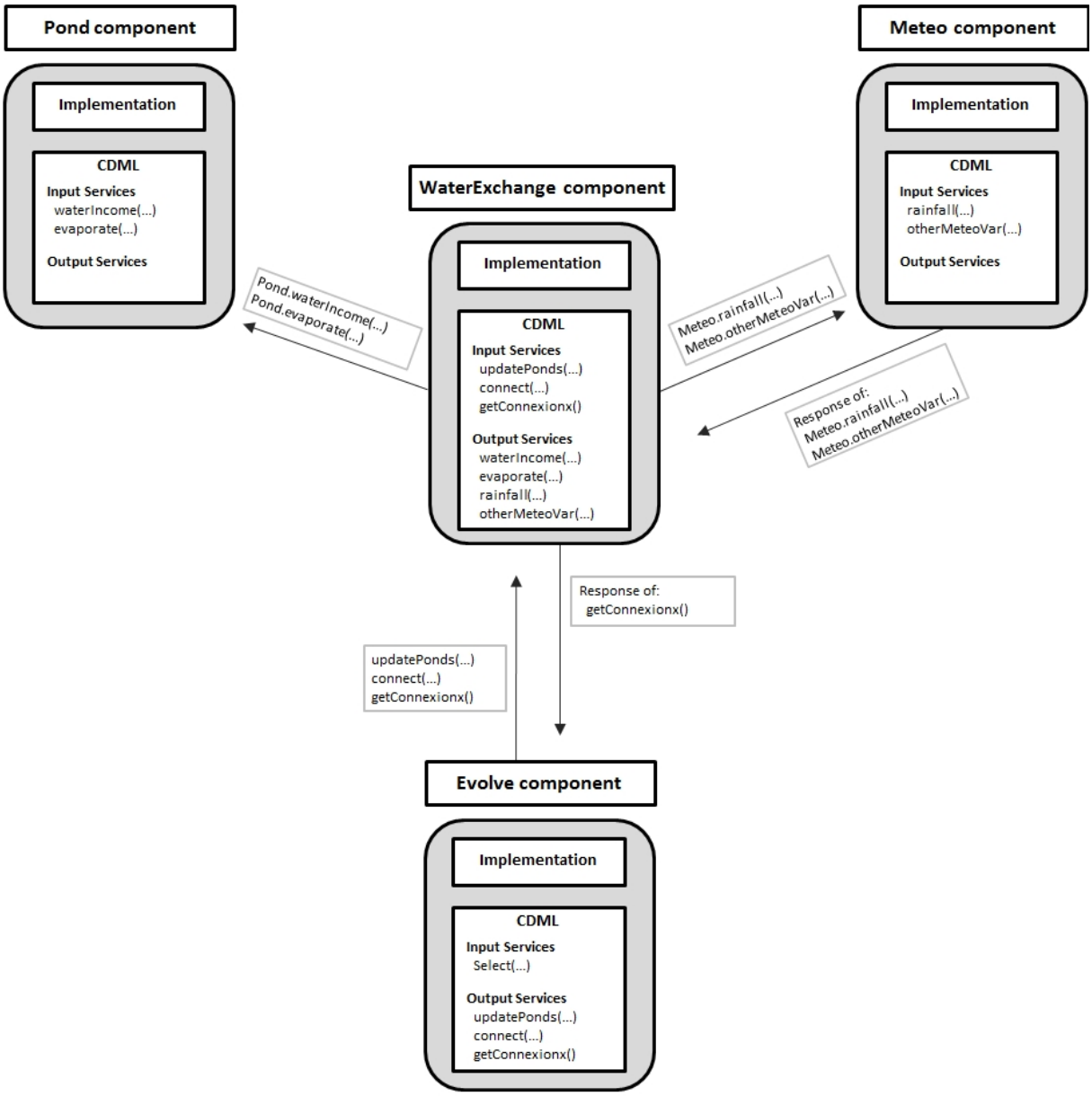

Figure 8.7: The SON components of the simple pond dynamics model.

The Pond entity only needs two services : *waterIncome()* and *evaporate()*. The first uses rainfall to calculate the amount of water the pond receives, the second takes other meteorological variables (solar radiation, wind speed, etc...) to estimate evaporation. These data would be obtained from a Meteo entity which also provides two services: *rainfall()* and *otherMeteoVar()*. A *waterExchange* relation models how one pond entity updates its water budget given the meteorological data obtained from a meteo entity. The relation is described as a one to one interaction between a meteo and a pond. But when initializing a simulation, it is likely that many instances $pond_i$ of the Pond entity will be created, each with different shapes and locations. Typically, a series of calls to a *waterExchange.connect(meteo,*$pond_i$*)* is needed to establish a link between one instance of Meteo entity and every $pond_i$ entity through the *waterExchange* relation. The *evolve* scenario will be executed for every time step of the simulation. That scenario is based on a *select* statement that will apply the *updatePond()* service of the relation to all the entities that had previously been connected. In other words, the series of calls to a *connect()* statement creates an interaction graph between one meteo entity and many $pond_i$ entities, and once that graph is built, a call to *updatePond* on the relation is enough to update all the ponds present in that interaction graph. The purpose of the *select* statement is to provide a way to activate only a subset of the interaction graph. For example, one can imagine a selection based on spatial attributes that would call *updatePond()* on all the ponds located in a given area.

As shown in Figure 8.7, the pond dynamics model written in Ocelet is translated into a SON application with four components. Each component corresponds to an element in the model (i.e., Pond, Meteo, WaterExchange, Evolve) and embodies a Java implementation generated from the Ocelet specification code, a CDML file that exposes the input services, the output services and the component container that embodies all resources needed to adapt the implementation code to the runtime environment.

The initial state of the application is described in the deployment file World. It contains a description of the component instances and connections that have to be created to launch the application. A component instance is identified by the couple (name of the component, name of the instance). As shown in Figure 8.8, WaterExchange_1 is instantiated from the relation component WaterExchange and connected to the instances Pond_1 and Meteo_1, while Evolve_1 is instantiated from the scenario component Evolve and connected to WaterExchange_1. Of

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<world>

<connectTo id_src="ComponentsManager" type_dest="Pond" id_dest="Pond_1"/>
<connectTo id_src="ComponentsManager" type_dest="Meteo" id_dest="Meteo_1"/>
<connectTo id_src="ComponentsManager" type_dest="Evolve" id_dest="Evolve_1"/>
<connectTo id_src="ComponentsManager" type_dest="WaterExchange"
                                                id_dest="WaterExchange_1"/>

<connectTo id_src="WaterExchange_1" type_dest="Pond" id_dest="Pond_1"/>
<connectTo id_src="WaterExchange_1" type_dest="Meteo" id_dest="Meteo_1"/>

<connectTo id_src="Evolve_1" type_dest="WaterExchange"
                                                id_dest="WaterExchange_1"/>

</world>
```

Figure 8.8: Deployment description file of the simple pond dynamics model.

course, after that, other component instances can be created and connected with each other dynamically during the execution as explained in Section 7.3.

When creating many instances of the Pond component, the specific shape and location of each pond can be obtained from an existing GIS file (a shapefile for example). The initialising scenario of the model needs to access the source of data through a Datafacer to obtain the unique parameters of every Pond it creates. For the present example, the DHT module of SON plays in the runtime the role of a Datafacer, it allows accessing to external data sources and provides services that can be called by the different components. More precisely, the DHT module gives some parameter settings, like the name and location of the shapefile, and some metadata needed to access the right attributes from the file.

Dynamic deployment when extending an existing model

The simple pond dynamics model presented earlier can be made more realistic by improving the description of its parts (which can be of different types: Pond, Meteo, WaterExchange, Evolve) without changing the logic of the model. When studying the RVF problem, however, more processes are also to be considered, among which, the dynamics of pastures, the displacement of herds between ponds and grazing areas, the development of mosquito populations in the ponds, and the transmission of the RVF virus. From the point of view of livestock management,

it may be enough to know how the grazing areas and ponds are changing during the season. As disease surveillance by veterinary services are mainly based on farmer reports, farmers can only estimate an a posteriori risk of animals being infected near the ponds. The point of view of the entomologist, with a good understanding of mosquito population dynamics in the ponds, and that of the epidemiologist, with the knowledge of how the virus is transmitted, would be needed to better estimate this risk.

The inclusion of the mosquito populations within the previous simulation can be done as follows. Once the simple pond Ocelet model has been tested and considered satisfactory, new mosquito population entities at different stages of their life cycle (egg, larva, pupa and imago) can be added and pond entity can be augmented with services that interact with mosquito entities. Then, the SON infrastructure generates the new components corresponding to mosquito entities and updates the pond component to take into account the new services. After that, these components can be dynamically integrated in the runtime without stopping neither the execution nor the components which are not affected by the modifications. The only condition is that the updated pond entity/component would provide, in addition to the new services, the same services as the first version to make sure that it would seamlessly integrate and interact with the rest of the components.

Part IV:

Closing

Chapter 9

Conclusions and future works

Contents

This chapter presents the concluding remarks of this thesis along with perspectives for future works. At first, Section 9.1 gives a summary of our major contributions, and then presents current assumptions and limitations of the proposed methods and techniques. Subsequently, Section 9.2 discusses the future work directions and their perspectives.

9.1 Conclusions

The main goal of this thesis is to facilitate the development of component-based applications with a data-centric approach and within a service-oriented P2P architecture. To achieve this purpose, we have proposed: i) a formal language, called DDF (Data Dependency Formalism), to specify such applications; ii) an analysis method of DDF specification based on data-flow principles; iii) a runtime middleware, called SON (Shared-data Overlay Network), for developing and deploying component-based services within a P2P architecture. The principle characteristics, assumptions and limitations of these contributions are summarized in the following points.

The principle characteristics of our contributions are summarized as follow:

Related to DDF specification

- Allowing parts of the control logic (even if it is recursive) to be described conceptually separated from other parts by using the concept of rules;
- The user describes what is to be done rather than the details of how is to be done;
- From a DDF specification, we can construct an abstract representation (i.e., Data-Dependency Graph). This abstraction exposes the right level of detail to perform data-flow analyzes;
- From a single specification, multiple implementations can be synthesized by analyzing the corresponding Data-Dependency Graph.

Related to the analysis of the specification

- The proposed analysis helps to identify the dependencies between data and between the steps of the specification, in particular, "non-direct" dependencies that can be very hard to identify without a computer analysis;
- Managing such dependencies leads to direct improvement in the application's running time. For example, reducing the number of dependencies may help to perform several optimizations (e.g., in execution time or memory usage);
- Several algorithms have been proposed in the field of AGs and DFA to infer and compute a broad range of properties. Our proposed analysis method explores to use these algorithms in the context of component-based applications.

Related to SON middleware

- SON middleware extends the principles of SOA as well as CBSE to support building applications within a P2P architecture in an effortless and effective way;
- SON provides a component model hiding the management of the underlying network issues to relieve software developers from P2P low level complicated tasks;

- SON's user implements only the code corresponding to the declared services. Afterwards, a code generation tool generates all resources needed to adapt the implementation code to the P2P runtime environment;

- SON has been evaluated in the context of the STAMP project. The objective was to show how SON (especially the dynamic availability of services during runtime) is able to improve and enhance the effectiveness of application scenarios from the area of modelling environmental landscapes and their dynamics.

The following points summarize the assumptions and the limitations of our proposals:

Related to DDF specification

- Although the semantic equations of the DDF rules specify the value for each output datum, in order to actually compute this value, the values of any input data that are arguments of the defining semantic equation must first be available. Such dependency relations restrict the order in which data can be computed.

- An important requirement in DDF is that the semantic equation of a rule should not have side-effects, i.e., it should not access or change a datum in the system if this datum is not in the set of input data of the rule. The reason for this restriction is that semantic equations represent definitions of the data values, and not effects of an execution.

- The semantic equation of a rule has no side-effects, neither do function invocations. These two conditions imply that the DDF specification is non-procedural. A disadvantage of this type of specification is that exception handling cannot be guaranteed to work effectively, because the control flow is not provided in explicit instructions.

Related to the analysis of the specification

- In extreme cases, a datum can depend on itself; such a situation occurs in ill-defined specification or when a system contains a deadlock. To resolve this problem, we search for circularity in the Data-Dependency Graph of the system. This solution has been inspired

from the attribute grammar theory, and it is known in this theory that the circularity test increases exponentially. Fortunately, there are interesting approaches that can help to deal with this problem in polynomial time [Deransart et al., 1988].

- The theory of AGs and DFA provide a wide range of algorithms to perform various evaluation orders of data and to compute different properties. However, a reformulation of these algorithms is needed. In particular, when software is built with independent and reusable components. The reason is that the most standard approaches for data-flow analysis do not take into account modular structure, and takes as input an entire program treated as a homogeneous entity.

Related to SON middleware

- Non-functional requirements such as security, privacy, response time, recovery, etc. need to be considered at some points in the lifecycle of all software systems. In the actual release of SON middleware, such non-functional requirements have not been treated. A simple reason for this is that they were not the first objective of SON.

- Some performance limitations in SON middleware rise from the fact that it relays on a DHT. In fact, although a request can be routed to the node that maintains the desired content quickly and accurately, the placement of content is tightly controlled. This implies that the cost of maintaining the structured topology of the overlay might be high, especially in a very large network environment [Vu et al., 2010].

9.2 Future works and perspectives

With the increasing interest in using component-based principles in software engineering, every approach that may ease the development of component-based applications is valued. In this thesis, we have established the foundations for a formalism, an analysis approach and a runtime environment to facilitate building component-based applications within a service-oriented P2P architecture. This work provides the basis for further research possibilities and, of course, gives

rise to a number of development and enhancement tasks that need to be improved with future efforts, in particular, those related to the limitations and the assumptions presented above.

One of the future tasks that we plan to work on first is finalizing the automatization of DDF specification and analysis within SON middleware. In fact, we consider implementing a graphical user interface to assist SON's users during the specification of application behaviors. This graphical interface will also provide the possibility to verify or compute some application properties with pre-implemented data-flow analysis algorithms. In chapter 6, we have treated deadlock and dominance detection problems and given the associated algorithms. However, other analysis algorithms (inspired for example from the works of Parigot on Grammar-Flow Analysis) can be reformulated to be used. For instance, by analyzing the order of data evaluation, we will be able to determine formally which services in a system can be executed in a parallel or incremental way.

Afterwards, we plan to extend our formalism by program transformation mechanisms in order to optimize resource allocations (e.g., optimize CPU and memory usage by analyzing the lifetime of data, while taking into account their functional dependencies and redundancies) in large-scale data-centric applications, in particular, in the emerging Cloud Computing area, where data management has been receiving significant attention. Another perspective field where this future work (i.e., optimization of resource allocations) might be useful is the Green Computing. In fact, environmental protection and energy-aware resource management have become popular and important research topics at present [Hu et al.,]. In this direction, the Green Computing is emerging as an indispensable part in sustaining the practice of protecting the environment on both individual and collective levels.

Bibliography

[Abadi, 2009] Abadi, D. J. (2009). Data management in the cloud: Limitations and opportunities. *IEEE Data Eng. Bull.*, 32(1):3–12.

[Aberer and Despotovic, 2001] Aberer, K. and Despotovic, Z. (2001). Managing trust in a peer-2-peer information system. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 310–317, New York, NY, USA. ACM.

[Aho et al., 2006] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Prentice Hall, 2 edition.

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Ait Lahcen et al., 2009] Ait Lahcen, A., Degenne, P., Lo Seen, D., and Parigot, D. (2009). Developing a service-oriented component framework for a landscape modeling language. In *Proceedings of the 13th International Conference on Software Engineering and Applications(SEA'09)*, pages 178–185, Cambridge, Massachusetts, USA.

[Allen and Cocke, 1972] Allen, F. E. and Cocke, J. (1972). Graph Theoretic Constructs For Program Control Flow Analysis. Technical report, IBM T.J. Watson Research Center, Yorktown Heights, NY.

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6:213–249.

[Alur et al., 2005] Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., and Yannakakis, M. (2005). Analysis of recursive state machines. *ACM Trans. Program. Lang. Syst.*, 27:786–818.

[Alvaro et al., 2010] Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. (2010). Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 223–236, New York, NY, USA. ACM.

[Andersen et al., 2001] Andersen, D., Balakrishnan, H., Kaashoek, F., and Morris, R. (2001). Resilient overlay networks. *SIGOPS Oper. Syst. Rev.*, 35:131–145.

[Arad and Haridi, 2010] Arad, C. and Haridi, S. (2010). Kompics: a message-passing component model for building distributed systems.

[Atkinson et al., 2008] Atkinson, C., Bostan, P., Brenner, D., Falcone, G., Gutheil, M., Hummel, O., Juhasz, M., and Stoll, D. (2008). Modeling components and component-based systems in kobra. In Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 54–84. Springer Berlin / Heidelberg.

[Attali, 1988] Attali, I. (1988). Compiling typol with attribute grammars. In *Programming Languages Implementation and Logic Programming*, volume 348, pages 252–272. Springer-Verlag. Orléans.

[Bah et al., 2006] Bah, A., Touré, I., Page, C. L., Ickowicz, A., and Diop, A. (2006). An agent-based model to understand the multiple uses of land and resources around drillings in sahel. *Mathematical and Computer Modelling*, 44(5-6):513–534.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. The MIT Press.

[Balakrishnan et al., 2003] Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., and Stoica, I. (2003). Looking up data in p2p systems. *Commun. ACM*, 46:43–48.

[Barkai, 2002] Barkai, D. (2002). *Peer-To-Peer Computing: Technologies for Sharing and Collaborating on the Net*. Engineer-To-Engineer. Intel Press.

[Batagelj and Mrvar, 1998] Batagelj, V. and Mrvar, A. (1998). PAJEK - Program for large network analysis. *Connections*, 21(2):47–57.

[Baumeister et al., 2006] Baumeister, H., Hacklinger, F., Hennicker, R., Knapp, A., and Wirsing, M. (2006). A component model for architectural programming. *Electronic Notes in Theoretical Computer Science*, 160(0):75 – 96.

[Baumeister et al., 2004] Baumeister, R. F., Zhang, L., and Vohs, K. D. (2004). Gossip as Cultural Learning. *Review of General Psychology*, 8(2):111–121.

[Benedikt et al., 2001] Benedikt, M., Godefroid, P., and Reps, T. W. (2001). Model checking of unrestricted hierarchical state machines. In *Proceedings of the 28th International Colloquium on Automata, Languages and Programming,*, ICALP '01, pages 652–666, London, UK. Springer-Verlag.

[Bisignano et al., 2003] Bisignano, M., Calvagna, A., Modica, G., and Tomarchio, O. (2003). Expeerience: a jxta middleware for mobile ad-hoc networks. In *Peer-to-Peer Computing, 2003. (P2P 2003). Proceedings. Third International Conference on*, pages 214 – 215.

[Börger, 2000] Börger, E. (2000). *Architecture design and validation methods*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Borshchev and Filippov, 2004] Borshchev, A. and Filippov, A. (2004). From System Dynamics and Discrete Event to Practical Agent Based Modeling: Reasons, Techniques, Tools. In *Proceedings of the 22nd International Conference of the System Dynamics Society*, pages 24–29.

[Bousquet et al., 1998] Bousquet, F., Bakam, I., Proton, H., and Page, C. L. (1998). Cormas: Common-pool resources and multi-agent systems. In *Proceedings of the 11th International Conference on Industrial and Engineering Applications of Artificial In telligence and Expert Systems: Tasks and Methods in Applied Artificial Intelligence*, IEA/AIE '98, pages 826–837, London, UK, UK. Springer-Verlag.

[Bousquet and Le Page, 2004] Bousquet, F. and Le Page, C. (2004). Multi-agent simulations and ecosystem management: A review. *Ecological Modelling*, 176(3-4):313–332.

[Boyland, 1996] Boyland, J. T. (1996). Conditional attribute grammars. *ACM Trans. Program. Lang. Syst.*, 18(1):73–108.

[Brown et al., 2005] Brown, D. G., Riolo, R., Robinson, D. T., North, M., and R, W. (2005). Spatial process and data models: Toward integration of agent-based models and gis. *Journal of Geographical Systems*, pages 25–47.

[Bruneton et al., 2006] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., and Stefani, J.-B. (2006). The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284.

[Buford et al., 2009] Buford, J., Yu, H., and Lua, E. (2009). *P2P Networking and Applications*. Morgan Kaufmann Series in Networking. Elsevier/Morgan Kaufmann.

[Buford and Yu, 2010] Buford, J. F. and Yu, H. (2010). Peer-to-peer networking and applications: Synopsis and research directions. In Shen, X., Yu, H., Buford, J., and Akon, M., editors, *Handbook of Peer-to-Peer Networking*, pages 3–45. Springer US.

[Bulej et al., 2008] Bulej, L., Bures, T., Coupaye, T., Decky, M., Jezek, P., Parízek, P., Plasil, F., Poch, T., Rivierre, N., Sery, O., and Tuma, P. (2008). Cocome in fractal. In Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 357–387. Springer Berlin / Heidelberg.

[Bures et al., 2008] Bures, T., Decky, M., Hnetynka, P., Kofron, J., Parízek, P., Plasil, F., Poch, T., Sery, O., and Tuma, P. (2008). Cocome in sofa. In Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 388–417. Springer Berlin / Heidelberg.

[Bures et al., 2006] Bures, T., Hnetynka, P., and Plasil, F. (2006). Sofa 2.0: Balancing advanced features in a hierarchical component model. In *Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, SERA '06, pages 40–48, Washington, DC, USA. IEEE Computer Society.

[Burkart and Steffen, 1992] Burkart, O. and Steffen, B. (1992). Model checking for context-free processes. In Cleaveland, W., editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 123–137. Springer Berlin / Heidelberg.

[Burkart and Steffen, 1994] Burkart, O. and Steffen, B. (1994). Pushdown processes: Parallel composition and model checking. In Jonsson, B. and Parrow, J., editors, *CONCUR '94: Concurrency Theory*, volume 836 of *Lecture Notes in Computer Science*, pages 98–113. Springer Berlin / Heidelberg.

[Burkart and Steffen, 1997] Burkart, O. and Steffen, B. (1997). Model checking the full modal mu-calculus for infinite sequential processes. In Degano, P., Gorrieri, R., and Marchetti-Spaccamela, A., editors, *Automata, Languages and Programming*, volume 1256 of *Lecture Notes in Computer Science*, pages 419–429. Springer Berlin / Heidelberg.

[Burrough et al., 1996] Burrough, P., Frank, A., and Foundation, E. S. (1996). *Geographic Objects With Indeterminate Boundaries*. GISDATA Series. Taylor & Francis.

[Burrough and Mcdonnell, 1998] Burrough, P. A. and Mcdonnell, R. A. (1998). *Principles of Geographical Information Systems*. Oxford University Press, USA.

[Cain et al., 2008] Cain, A., Chen, T. Y., Grant, D. D., Kuo, F.-C., and Schneider, J.-G. (2008). An object oriented approach towards dynamic data flow analysis (short paper). In *Proceedings of the 2008 The Eighth International Conference on Quality Software*, QSIC '08, pages 163–168, Washington, DC, USA. IEEE Computer Society.

[Cambridge Academic Content Dictionary, 2008] Cambridge Academic Content Dictionary (2008). *Cambridge Academic Content Dictionary*. Cambridge University Press.

[Chen et al., 2008] Chen, Z., Hannousse, A., Van Hung, D., Knoll, I., Li, X., Liu, Z., Liu, Y., Nan, Q., Okika, J., Ravn, A., Stolz, V., Yang, L., and Zhan, N. (2008). Modelling with relational calculus of object and component systems - rcos. In Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 116–145. Springer Berlin / Heidelberg.

[Chomsky, 1956] Chomsky, N. (1956). Three models for the description of language. *Information Theory, IRE Transactions on*, 2(3):113–124.

[Claramunt and Thériault, 1995] Claramunt, C. and Thériault, M. (1995). Managing Time in GIS: An Event-Oriented Approach. In *Proceedings of the International Workshop on Temporal Databases: Recent Advances in Temporal Databases*, pages 23–42, London, UK, UK. Springer-Verlag.

[Correnson, 2000] Correnson, L. (2000). *SÉmantique Equationnelle*. Phd thesis, Ecole Polytechnique.

[Correnson et al., 1999] Correnson, L., Duris, E., Parigot, D., and Roussel, G. (1999). Equational semantics. In *Proceedings of the 6th International Symposium on Static Analysis*, SAS '99, pages 264–283, London, UK, UK. Springer-Verlag.

[Couclelis, 1992] Couclelis, H. (1992). People manipulate objects (but cultivate fields): Beyond the raster-vector debate in gis. In *Proceedings of the International Conference GIS - From Space to Territory: Theories and Methods of Spatio-Temporal Reasoning on Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, pages 65–77, London, UK, UK. Springer-Verlag.

[Coulson et al., 2008] Coulson, G., Blair, G., Grace, P., Taiani, F., Joolia, A., Lee, K., Ueyama, J., and Sivaharan, T. (2008). A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26:1:1–1:42.

[Cova and Goodchild, 2002] Cova, T. J. and Goodchild, M. F. (2002). Extending geographical representation to include fields of spatial objects. *International Journal of Geographical Information Science*, pages 509–532.

[Degenne et al., 2010] Degenne, P., Ait Lahcen, A., Curé, O., Forax, R., Parigot, D., and Lo Seen, D. (2010). Modelling with behavioural graphs. Do you speak Ocelet? In *Proceedings of the International Congress on Environmental Modelling and Software (iEMSs)*, Ottawa, Ontario, Canada.

[Degenne et al., 2009] Degenne, P., Lo Seen, D., Parigot, D., Forax, R., Tran, A., Ait Lahcen, A., Curé, O., and Jeansoulin, R. (2009). Design of a domain specific language for modelling processes in landscapes. *Ecological Modelling*, 220(24):3527 – 3535.

[Demchak et al., 2008] Demchak, B., Ermagan, V., Farcas, E., Huang, T.-j., Kruger, I., and Menarini, M. (2008). A rich services approach to cocome. In Rausch, A., Reussner, R., Mirandola, R., and Plasil, F., editors, *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*, pages 85–115. Springer Berlin / Heidelberg.

[Demers et al., 1987] Demers, A., Greene, D., Hauser, C., Irish, W., Larson, J., Shenker, S., Sturgis, H., Swinehart, D., and Terry, D. (1987). Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA. ACM.

[Deransart and Jourdan, 1990] Deransart, P. and Jourdan, M., editors (1990). *Attribute Grammars and their Applications, International Conference WAGA, Paris, France, September 19-21, 1990, Proceedings*, volume 461 of *Lecture Notes in Computer Science*. Springer.

[Deransart et al., 1988] Deransart, P., Jourdan, M., and Lorho, B. (1988). *Attribute grammars: definitions, systems and bibliography*. Springer-Verlag, Inc., New York, NY, USA.

[Devillers and Jeansoulin, 2006] Devillers, R. and Jeansoulin, R. (2006). *Fundamentals of Spatial Data Quality*. Geographical Information Systems Series. Iste.

[Draidi et al., 2011a] Draidi, F., Pacitti, E., and Kemme, B. (2011a). A P2P Recommendation System for Large-scale Data Sharing. *Transactions on Large-Scale Data- and Knowledge Centered Systems*, 6790(3):87–116.

[Draidi et al., 2011b] Draidi, F., Pacitti, E., Parigot, D., and Verger, G. (2011b). P2prec: a social-based p2p recommendation system. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 2593–2596, New York, NY, USA. ACM.

[Duboz et al., 2003] Duboz, R., Ramat, E., and Preux, P. (2003). Scale transfer modeling: using emergent computation for coupling an ordinary differential equation system with a reactive agent model. *Syst. Anal. Model. Simul.*, 43(6):793–814.

[Duris, 1998] Duris, É. (1998). *Contribution aux relations entre les grammaires attribuées et la programmation fonctionnelle*. Phd thesis, Université de Marne la Vallée.

[Erl, 2007] Erl, T. (2007). *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

[Esparza et al., 2000] Esparza, J., Hansel, D., Rossmanith, P., and Schwoon, S. (2000). Efficient algorithms for model checking pushdown systems. In Emerson, E. and Sistla, A., editors, *Computer Aided Verification*, volume 1855 of *Lecture Notes in Computer Science*, pages 232–247. Springer Berlin / Heidelberg.

[Eugster et al., 2004] Eugster, P. T., Guerraoui, R., Kermarrec, A.-M., and Massoulieacute;, L. (2004). Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67.

[Fall and Fall, 2001] Fall, A. and Fall, J. (2001). A domain-specific language for models of landscape dynamics. *Ecological Modelling*, 141(1-3):1–18+.

[Farrow, 1986] Farrow, R. (1986). Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, SIGPLAN '86, pages 85–98, New York, NY, USA. ACM.

[Foster, 2004] Foster, E. K. (2004). Research on gossip: Taxonomy, methods, and future directions. *Foster*, 8(2):78–99.

[Freenet, 2012] Freenet (2012). https://freenetproject.org/.

[Fuller and Sarkar, 2006] Fuller, T. and Sarkar, S. (2006). Lqgraph: A software package for optimizing connectivity in conservation planning. *Environmental Modelling & Software*, 21(5):750–755.

[Gaff et al., 2007] Gaff, H. D., Hartley, D. M., and Leahy, N. P. (2007). An epidemiological model of Rift Valley fever. *Electronic Journal of Differential Equations*, 115:1–12.

[Galuba and Girdzijauskas, 2009] Galuba, W. and Girdzijauskas, S. (2009). Peer to peer overlay networks: Structure, routing and maintenance. In Liu, L. and Özsu, M. T., editors, *Encyclopedia of Database Systems*, pages 2056–2061. Springer US.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of reusable object-oriented software*. Addison-Wesley Publishing.

[Garousi et al., 2005] Garousi, V., Briand, L., and Labiche, Y. (2005). Control flow analysis of uml 2.0 sequence diagrams. In Hartman, A. and Kreische, D., editors, *Model Driven Architecture - Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 160–174. Springer Berlin / Heidelberg.

[Gartner Research Group, 2001] Gartner Research Group (2001). The emergence of distributed content management and peer-to-peer content networks.

[Gaucherel et al., 2006] Gaucherel, C., Giboire, N., Viaud, V., Houet, T., Baudry, J., and Burel, F. (2006). A domain-specific language for patchy landscape modelling: The brittany agricultural mosaic as a case study. *Ecological Modelling*, 194(1-3):233–243.

[Giesecke et al., 2005] Giesecke, S., Warns, T., and Hasselbring, W. (2005). Availability simulation of peer-to-peer architectural styles. *SIGSOFT Softw. Eng. Notes*, 30(4).

[Gnutella, 2012] Gnutella (2012). www.gnutella.com.

[Goodchild, 1992] Goodchild, M. F. (1992). Geographical data modeling. *Comput. Geosci.*, 18(4):401–408.

[Gorton, 2011] Gorton, I. (2011). *Essential Software Architecture (2nd ed.)*. Springer.

[Govindarajan et al., 1992] Govindarajan, R., Yu, S., and Lakshmanan, V. S. (1992). Attempting guards in parallel: A data flow approach to execute generalized guarded commands. *International Journal of Parallel Programming*, 21:225–268.

[Gu et al., 2004] Gu, X., Nahrstedt, K., and Yu, B. (2004). Spidernet: an integrated peer-to-peer service composition framework. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 110 – 119.

[GWT, 2007] GWT (2007). Google Web Toolkit - Build AJAX apps in the Java language. http://code.google.com/webtoolkit/.

[Hill, 2006] Hill, L. L. (2006). *Georeferencing : the geographic associations of information*. Digital libraries and electronic publishing. MIT Press, Cambridge, Mass.

[Hoare and Jifeng, 1998] Hoare, C. and Jifeng, H. (1998). *Unifying theories of programming*. Prentice Hall series in computer science. Prentice Hall.

[Hofmann and Beaumont, 2005] Hofmann, M. and Beaumont, L. R. (2005). *Content Networking: Architecture, Protocols, and Practice (The Morgan Kaufmann Series in Networking)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Hu et al.,] Hu, J., Deng, J., and Wu, J. A green private cloud architecture with global collaboration. *Telecommunication Systems*, pages 1–11.

[Huhns and Singh, 2005] Huhns, M. N. and Singh, M. P. (2005). Service-oriented computing : Key concepts and principles. *IEEE Internet Computing*, 9:75–81.

[IEEE, 2000] IEEE, A. (2000). Ieee std 1471-2000, recommended practice for architectural description of software-intensive systems. Technical report, IEEE.

[Jaffar-ur Rehman et al., 2007] Jaffar-ur Rehman, M., Jabeen, F., Bertolino, A., and Polini, A. (2007). Testing software components for integration: a survey of issues and techniques: Research articles. *Softw. Test. Verif. Reliab.*, 17(2):95–133.

[Jelasity, 2011] Jelasity, M. (2011). Gossip. In Di Marzo Serugendo, G., Gleizes, M.-P., and Karageorgos, A., editors, *Self-organising Software*, Natural Computing Series, pages 139–162. Springer Berlin Heidelberg.

[Jelasity et al., 2007] Jelasity, M., Voulgaris, S., Guerraoui, R., Kermarrec, A.-M., and van Steen, M. (2007). Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25.

[Jourdan and Parigot, 1990] Jourdan, M. and Parigot, D. (1990). Techniques for improving grammar flow analysis. In *Proceedings of the third European symposium on programming on ESOP '90*, pages 240–255, New York, NY, USA. Springer-Verlag New York, Inc.

[Kaisler, 2005] Kaisler, S. (2005). *Software Paradigms*. Wiley.

[Kam and Ullman, 1976] Kam, J. B. and Ullman, J. D. (1976). Global data flow analysis and iterative algorithms. *J. ACM*, 23:158–171.

[Kermarrec and van Steen, 2007] Kermarrec, A.-M. and van Steen, M. (2007). Gossiping in distributed systems. *SIGOPS Oper. Syst. Rev.*, 41(5):2–7.

[Killian et al., 2007] Killian, C. E., Anderson, J. W., Braud, R., Jhala, R., and Vahdat, A. M. (2007). Mace: language support for building distributed systems. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 179–188, New York, NY, USA. ACM.

[Knapp et al., 2008] Knapp, A., Janisch, S., Hennicker, R., Clark, A., Gilmore, S., Hacklinger, F., Baumeister, H., and Wirsing, M. (2008). The common component modeling example. chapter Modelling the CoCoME with the Java/A Component Model, pages 207–237. Springer-Verlag, Berlin, Heidelberg.

[Knuth, 1968] Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145. Correction: sl Mathematical Systems Theory 5, 1, pp. 95-96 (March 1971).

[Kruchten, 1998] Kruchten, P. (1998). Modeling Component Systems with the Unified Modeling Language. In *International Workshop on Component-Based Software Engineering*.

[Kwok, 2011] Kwok, Y. (2011). *Peer-To-Peer Computing: Applications, Architecture, Protocols, and Challenges*. Computational Science. Taylor & Francis.

[Langran, 1992] Langran, G. (1992). Time in geographic information systems. *Geocarto International*, 7(2):40.

[Launchbury and Sheard, 1995] Launchbury, J. and Sheard, T. (1995). Warm fusion: Deriving build-cata's from recursive definitions. In *on Func. Prog. Languages and Computer Architecture FPCA'95*, FPCA'95, pages 314–323. ACM Press.

[Lee et al., 2007] Lee, J., Lee, H., Kang, S., Kim, S. M., and Song, J. (2007). Ciss: An efficient object clustering framework for dht-based peer-to-peer applications. *Comput. Netw.*, 51(4):1072–1094.

[Leeb, 1996] Leeb, A. (1996). *A Flexible Object Architecture for Component Software*. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science.

[Lin et al., 2005] Lin, S., Pan, A., Guo, R., and Zhang, Z. (2005). Simulating large-scale p2p systems with the wids toolkit. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 415 – 424.

[Lin et al., 2011] Lin, S., Taïani, F., Bertier, M., Blair, G., and Kermarrec, A.-M. (2011). Transparent componentisation: high-level (re)configurable programming for evolving distributed systems. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 203–208, New York, NY, USA. ACM.

[Linard et al., 2009] Linard, C., Ponçon, N., Fontenille, D., and Lambin, E. F. (2009). A multi-agent simulation to assess the risk of malaria re-emergence in southern france. *Ecological Modelling*, 220(2):160 – 174.

[Liu et al., 2006] Liu, J., He, J., and Liu, Z. (2006). A strategy for service realization in service-oriented design. *Science in China Series F: Information Sciences*, 49:864–884.

[Liu and Antonopoulos, 2010] Liu, L. and Antonopoulos, N. (2010). From client-server to p2p networking. In Shen, X., Yu, H., Buford, J., and Akon, M., editors, *Handbook of Peer-to-Peer Networking*, pages 71–89. Springer US.

[Loo et al., 2005] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. (2005). Implementing declarative overlays. In *SOSP*, pages 75–90. ACM.

[Lotka, 1925] Lotka, A. (1925). *Elements of Physical Biology*, page 460. Williams and Wilkins, Baltimore, MD.

[Manolakos et al., 2001] Manolakos, E. S., Galatopoullos, D. G., and Funk, A. (2001). Component-based peer-to-peer distributed processing in heterogeneous networks using java ports. In *Proceedings of the IEEE International Symposium on Network Computing and Applications (NCA'01)*, pages 234–, Washington, DC, USA. IEEE Computer Society.

[Mauthe and Hutchison, 2003] Mauthe, A. and Hutchison, D. (2003). Peer-to-peer computing: Systems, concepts and characteristics. *Praxis der Informationsverarbeitung und Kommunikation*, 26(2):60–64.

[Maxwell and Costanza, 1997] Maxwell, T. and Costanza, R. (1997). A language for modular spatio-temporal simulation. *Ecological Modelling*, 103(2-3):105–113.

[McKinley et al., 2004] McKinley, P. K., Masoud, S. S., Kasten., E. P., and Cheng, B. H. C. (2004). Composing adaptive software. *IEEE Computer*, 37(7):56–64.

[Mernik et al., 2005] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

[Milner, 1980] Milner, R. (1980). *A calculus of communicating systems*. Lecture notes in computer science. Springer-Verlag.

[Murray, 2003] Murray, J. (2003). *Mathematical Biology: I: An introduction*. Springer.

[Napster, 2012] Napster (2012). http://www.napster.com.

[Ng et al., 2002] Ng, W. S., Ooi, B. C., Beng, N., Ooi, C., and lee Tan, K. (2002). Bestpeer: A self-configurable peer-to-peer system. In *Proceedings of the 18th International Conference on Data Engineering*, ICDE'02, pages 272–, Washington, DC, USA. IEEE Computer Society.

[OMG, 2002] OMG (2002). Corba component model 3.0 specification. Technical Report Version 3.0, Object Management Group.

[OMG, 2006] OMG (2006). Corba component model 4.0 specification. Technical Report Version 4.0, Object Management Group.

[OMG, 2007] OMG (2007). OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2. Technical report.

[Onoue et al., 1997] Onoue, Y., Hu, Z., Takeichi, M., and Iwasaki, H. (1997). A calculational fusion system hylo. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 76–106, London, UK, UK. Chapman & Hall, Ltd.

[Oracle, 2012] Oracle (2012). Enterprise javabeans technology. http://www.oracle.com/technetwork/java/javaee/ejb/index.html.

[Otero et al., 2008] Otero, M., Schweigmann, N., and Solari, H. (2008). A stochastic spatial dynamical model for aedes aegypti. *Bull Math Biol*, 70(5):1297–325.

[P2Prec, 2011] P2Prec (2011). Social-based P2P recommendation system (P2Prec). http://www-sop.inria.fr/teams/zenith/P2Prec.

[Paakki, 1995] Paakki, J. (1995). Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Comput. Surv.*, 27(2):196–255.

[Panda et al., 2007] Panda, D., Rahman, R., and Lane, D. (2007). *EJB 3 in Action*. Manning Publications Co., Greenwich, CT, USA.

[Papazoglou and Heuvel, 2003] Papazoglou, M. P. and Heuvel, W.-J. (2003). Service-oriented computing. *Communications of the ACM*, 46(10):25–28.

[Parent et al., 2006] Parent, C., Spaccapietra, S., and Zimányi, E. (2006). *Conceptual Modeling for Traditional and Spatio-Temporal Applications: The MADS Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

[Parigot et al., 1996] Parigot, D., Roussel, G., Jourdan, M., and Duris, E. (1996). Dynamic attribute grammars. In *Int. Symp. on Progr. Languages, Implementations, Logics and Programs (PLILP'96)*, volume 1140, pages 122–136. Springers.

[Peuquet, 1994] Peuquet, D. J. (1994). It's about Time: A Conceptual Framework for the Representation of Temporal Dynamics in Geographic Information Systems. *Annals of the Association of American Geographers*, 84(3):441–461.

[Peuquet, 2001] Peuquet, D. J. (2001). Making space for time: Issues in space-time data representation. *Geoinformatica*, 5(1):11–32.

[Plásil et al., 1998] Plásil, F., Bálek, D., and Janecek, R. (1998). Sofa/dcup: Architecture for component trading and dynamic updating. In *Proceedings of the International Conference on Configurable Distributed Systems*, CDS '98, pages 43–, Washington, DC, USA. IEEE Computer Society.

[Puntigam, 2003] Puntigam, F. (2003). State information in statically checked interfaces. In *Eighth International Workshop on Component-Oriented Programming*, Darmstadt, Germany.

[Ranganathan et al., 2002] Ranganathan, K., Iamnitchi, A., and Foster, I. (2002). Improving data availability through dynamic model-driven replication in large peer-to-peer communities. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '02, pages 376–, Washington, DC, USA. IEEE Computer Society.

[Ratze et al., 2007] Ratze, C., Gillet, F., Muller, J., and Stoffel, K. (2007). Simulation modelling of ecological hierarchies in constructive dynamical systems. *Ecological Complexity*, 4(1-2):13–25.

[Rausch et al., 2008] Rausch, A., Reussner, R., Mirandola, R., and Plasil, F. (2008). *The Common Component Modeling Example: Comparing Software Component Models*. Springer Publishing Company, Incorporated, 1st edition.

[Reussner, 2002] Reussner, R. (2002). Counter-constrained finite state machines: A new model for component protocols with resource-dependencies. In Grosky, W. and Plasil, F., editors, *SOFSEM 2002: Theory and Practice of Informatics*, volume 2540 of *Lecture Notes in Computer Science*, pages 20–40. Springer Berlin / Heidelberg.

[Rhea et al., 2004] Rhea, S., Geels, D., Roscoe, T., and Kubiatowicz, J. (2004). Handling churn in a dht. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 10–10, Berkeley, CA, USA. USENIX Association.

[Ripeanu et al., 2002] Ripeanu, M., Foster, I., and Iamnitchi, A. (2002). Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6:2002.

[Saura and Torné, 2009] Saura, S. and Torné, J. (2009). Conefor sensinode 2.2: A software package for quantifying the importance of habitat patches for landscape connectivity. *Environmental Modelling & Software*, 24(1):135–139.

[Schollmeier, 2001] Schollmeier, R. (2001). A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the First International Conference on Peer-to-Peer Computing*, P2P '01, pages 101–, Washington, DC, USA. IEEE Computer Society.

[Sheard, 1997] Sheard, T. (1997). A type-directed, on-line, partial evaluator for a polymorphic language. In *Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '97, pages 22–35, New York, NY, USA. ACM.

[Silberschatz et al., 2008] Silberschatz, A., Galvin, P. B., and Gagne, G. (2008). *Operating System Concepts*. Wiley Publishing, 8th edition.

[SON, 2011] SON (2011). Shared-data Overlay Network (SON) infrastructure. http://www-sop.inria.fr/teams/zenith/SON.

[Srinivasan and Wolfe, 1992] Srinivasan, H. and Wolfe, M. (1992). Analyzing programs with explicit parallelism. In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 405–419, London, UK, UK. Springer-Verlag.

[Stanley, 2001] Stanley, R. (2001). *Enumerative combinatorics*. Number vol. 2 in Cambridge studies in advanced mathematics. Cambridge University Press.

[Stoica et al., 2001] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA. ACM.

[Swierstra and Vogt, 1991] Swierstra, S. D. and Vogt, H. (1991). Higher order attribute grammars. In *Attribute Grammars, Applications and Systems*, pages 256–296.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software : Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition.

[Taylor and Harrison, 2004] Taylor, I. and Harrison, A. (2004). *From P2P to Web Services and Grids: Peers in a Client/Server World*. Computer Communications and Networks. Springer.

[Taylor et al., 2009] Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software Architecture: Foundations, Theory, and Practice*. Wiley Publishing.

[The Eclipse Foundation, 2003] The Eclipse Foundation (2003). *Eclipse Platform Technical Overview*.

[The OSGi Alliance, 2007] The OSGi Alliance (2007). OSGi service platform core specification. http://www.osgi.org/Specifications.

[The OSGi Alliance, 2012] The OSGi Alliance (2012). OSGi enterprise specification. http://www.osgi.org/Specifications.

[Torrens and Benenson, 2005] Torrens, P. M. and Benenson, I. (2005). Geographic automata systems. *International Journal of Geographical Information Science*, 19:4–385.

[Tran and Raffy, 2006] Tran, A. and Raffy, M. (2006). On the dynamics of dengue epidemics from large-scale information. *Theoretical Population Biology*, 69(1):3 – 12.

[Tyson et al., 2008] Tyson, G., Mauthe, A., Plagemann, T., and El-khatib, Y. (2008). Juno: Reconfigurable middleware for heterogeneous content networking. In *In Proc. 5th Intl. Workshop on Next Generation Networking Middleware (NGNM), Samos Islands, Greece*.

[Uhrmacher and Schattenberg, 1998] Uhrmacher, A. M. and Schattenberg, B. (1998). Agents in discrete event simulation. In Bargiela, A. and Kerckhoffs, E., editors, *10TH European Simulation Symposium "Simulation in Industry – Simulation Technology: Science and Art" (ESS'98)*, pages 129–136, Nottingham, UK. The Society for Computer Simulation International (SCS), SCS Publications, Ghent.

[Volterra, 1926] Volterra, V. (1926). Fluctuations in the abundance of a species considered mathematically. *Nature*, 118:558–560.

[Voulgaris et al., 2005] Voulgaris, S., Gavidia, D., and van Steen, M. (2005). Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13:197–217.

[Vu et al., 2010] Vu, Q. H., Lupu, M., Ooi, B. C., Vu, Q. H., Lupu, M., and Ooi, B. C. (2010). Architecture of peer-to-peer systems. In *Peer-to-Peer Computing*, pages 11–37. Springer Berlin Heidelberg.

[W3C, 2004] W3C (2004). Web Services Architecture, W3C Working Group Note. http://www.w3.org/TR/ws-arch.

[Wachowicz and Wachowiez, 1999] Wachowicz, J. M. and Wachowiez, M. (1999). *Object-Oriented Design for Temporal GIS*. Taylor & Francis, Inc., Bristol, PA, USA.

[Wainer and Giambiasi, 2005] Wainer, G. A. and Giambiasi, N. (2005). Cell-DEVS/GDEVS for Complex Continuous Systems. *Simulation*, 81(2):137–151.

[Wang and Qian, 2005] Wang, A. and Qian, K. (2005). *Component-oriented programming*. John Wiley & Sons.

[Wilson, 2002] Wilson, B. J. (2002). *JXTA*. New Riders.

[Worboys, 1994] Worboys, M. F. (1994). A Unified Model for Spatial and Temporal Information. *The Computer Journal*, 37(1):26–34.

[Wu, 1999] Wu, J. (1999). Hierarchy and scaling: Extrapolating information along a scaling ladder. *Canadian Journal of Remote Sensing*, 25:367–380.

[Yang and Papazoglou, 2004] Yang, J. and Papazoglou, M. P. (2004). Service components for managing the life-cycle of service compositions. *Inf. Syst.*, 29:97–125.

[Yang et al., 2009] Yang, X., Huang, J., and Gong, Y. (2009). Static data flow analysis and anomalies detection for bpel. In *Test and Measurement, 2009. ICTM '09. International Conference on*, volume 2, pages 18 –21.

[Yuan, 1999] Yuan, M. (1999). Use of a Three-Domain Repesentation to Enhance GIS Support for Complex Spatiotemporal Queries. *Transactions in GIS*, 3(2):137–159.

[Zeigler, 1984] Zeigler, B. P. (1984). *Theory of Modelling and Simulation*. Krieger Publishing Co., Inc., Melbourne, FL, USA.

[Zhou and Lee, 2006] Zhou, Y. and Lee, E. A. (2006). A causality interface for deadlock analysis in dataflow. In *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, EMSOFT '06, pages 44–52, New York, NY, USA. ACM.

Abbreviations

AG	Attribute Grammar
CA	Cellular Automata
CBSE	Component-based Software Engineering
CCM	CORBA Component Model
CD	Computed Data
CDML	Component Description Meta Language
CG	Component Generator
CM	Component Manager
CORBA	Common Object Request Broker Architecture
CSP	Communicating Sequential Processes
DAG	Dynamic Attributed Grammar
DDF	Data-Dependency Formalism
DDG	Data-Dependency Graph
DE	Discrete Event
DFA	Data-Flow Analysis
DHT	Distributed Hash Table
DNS	Domain Name Service
DSL	Domain-Specific Language
EJB	Enterprise JavaBeans

FSA	Finite-State Automaton
GIS	Geographic Information Systems
GUI	Graphical user interface
GWT	Google Web Toolkit
ID	Input Data
IDL	Interface Definition Language
JAR	Java ARchive
JVM	Java Virtual Machine
OD	Output Data
OOP	Object-Oriented Programming
P2P	Peer-to-Peer
RVF	Rift Valley Fever
SD	System Dynamics
SGT	Simple Georeferencing Tool
SOA	Service-Oriented Architecture
SON	Shared-data Overlay Network
STAMP	Spatial, Temporal And Multi-scale Primitives
UTP	Unifying Theories of Programming
UUID	Universally Unique Identifier