Université de la Méditerranée - Aix-Marseille II
Faculté des Sciences de Luminy

Contribution to the use of Constraint Programming for Finite Model Search in Artificial Intelligence

---

Contribution à l'utilisation de la Programmation par Contraintes pour la Recherche de Modèles Finis en Intelligence Artificielle

THÈSE

pour obtenir le grade de

docteur de l'université de la méditerranée
Discipline : Informatique

présentée et soutenue publiquement par

Mathias KLEINER

le 11 décembre 2007

Directeur de thèse : M. Laurent Henocque

Ecole Doctorale de Mathématiques et d'Informatique de Marseille - E.D. 184

Devant le Jury composé de :

M. Patrick Albert	Examinateur
M. John Domingue	Examinateur
M. Laurent Henocque	Directeur de thèse
M. Philippe Jégou	Examinateur
M^{me} Christine Solnon	Rapporteur
M. Markus Stumptner	Rapporteur

Année 2007

Remerciements

Je tiens tout d'abord à remercier Christine Solnon et Markus Stumptner de m'avoir fait l'honneur de rapporter sur cette thèse et d'avoir consacré de leur temps à la lecture du manuscrit. Je remercie également John Domingue et Philippe Jégou pour avoir accepté de faire partie du jury.
Je ne pourrai en quelques lignes exprimer toute ma gratitude à Laurent Henocque pour son soutien tout au long de ma thèse. Il m'a offert une grande liberté dans mes recherches tout en veillant à la qualité et à la diffusion de mon travail. Ses nombreux conseils, son aide et ses encouragements ont toujours été extrêmement précieux.
Je remercie Patrick Albert qui m'a donné l'opportunité de réaliser cette thèse. Il m'a dès le début accordé sa confiance et m'a souvent fourni un point de vue original et constructif sur mes recherches. Je remercie aussi Christian de Sainte Marie, qui m'a accompagné et soutenu dans la complexité d'un projet européen. Je remercie d'ailleurs les nombreuses personnes du projet DIP avec qui mes relations professionnelles furent aussi agréables qu'enrichissantes. Enfin je veux remercier les personnes de la société ILOG qui, par leur disponibilité et leur attention, m'ont permis de réaliser ma thèse dans d'excellentes conditions.
Je remercie également les membres de l'équipe InCA, permanents et temporaires, pour l'ambiance détendue et constructive qu'ils ont su mettre en place. Chacun à leur manière ils m'ont aidé au cours de ces dernières années, et même si je ne peux ici les citer tous, je leur en suis très reconnaissant. Je remercie particulièrement Philippe Jégou pour ses encouragements et son soutien, Nicolas Prcovic pour sa collaboration importante, ainsi que Cyril Terrioux pour sa disponibilité.
Ces années auraient été bien plus difficiles si je n'avais eu la chance de pouvoir compter sur mes amis de Marseille ou d'ailleurs. Je n'oserai les citer de peur d'en oublier, mais qu'ils sachent qu'ils ont tous contribué à ce manuscrit. Je tiens à remercier particulièrement Mathieu Estratat, qui m'a été d'un grand support en tant que collègue mais avant tout ami.
Je veux également exprimer chaleusement ma gratitude à mes parents ainsi qu'à ma soeur Nadia, dont le soutien et la confiance inconditionnels m'ont souvent réconforté.
Enfin, il est une personne qui m'a accompagné, supporté, soutenu et encouragé plus que personne durant toutes ces années. Gali, infiniment, merci.

Table of Contents

List of Figures

List of Tables

Résumé en français

1 Introduction

Cette thèse a pour objet d'étendre l'utilisation pratique de la programmation par contraintes (PC) en intelligence artificielle (IA). Plus précisément, nous contribuons à l'utilisation de la configuration, sous la forme de recherche énumérative de modèles finis pour les modèles objets contraints, dans des problèmes d'IA du premier ordre.
La programmation par contraintes est devenue un domaine d'intense activité en IA, notamment grâce à la notion de Problème de Satisfaction de Contraintes (CSP). Les CSP associent un modèle totalement déclaratif à des algorithmes de résolution qui ont montré leur efficacité dans de nombreux problèmes. En particulier, diverses méthodes ont été développées pour traiter l'explosion combinatoire inhérente à la recherche énumérative: heuristiques de variables et de valeurs, maintien de la consistance, réduction des domaines par propagation des contraintes, rechercher incomplète, etc.
Différentes extensions des CSPs ont également été proposées pour traiter des problèmes de logique du premier ordre. Cependant, leur utilisation pratique soulève de nombreux problèmes. Du point de vue de la modélisation, l'expression de contraintes en présence d'un nombre dynamique de variables (d'ensemble) est difficile. Du point de vue de la résolution, les théories du premier ordre entraînent une augmentation de la complexité, et l'emploi des méthodes utilisées pour les CSPs introduit des difficultés originales.
Les problèmes de configuration impliquent la création de structures complexes contenant un nombre dynamique de composants interconnectés. Les approches à base de PC, à travers des extensions des CSPs, apparaissent comme une solution potentielle adaptée, bien que soumise aux difficultés déjà évoquées. Les configurateurs orientés objet, qui combinent la logique des descriptions avec la PC, offrent donc de nombreux avantages. L'approche fournit en effet une grande liberté dans la déclaration de modèles et de contraintes, et profite de l'efficacité de la PC dans les méthodes de résolution.
Pourtant, malgré un potentiel intéressant, la configuration est rarement considérée comme une option pour des problèmes généraux d'IA impliquant du raisonnement symbolique. Ceci est peut-être du à sa traditionnelle utilisation industrielle, à l'expressivité réduite des configurateurs existants, ou à l'absence d'un formalisme commun pour représenter et résoudre les problèmes. En effet, la plupart des configurateurs ont leur propre formalisme. Celui-ci est généralement proche d'une méthode de résolution spécifique, et le contexte général de la configuration est souvent restreint (par exemple, la création dynamique de variables n'est pas toujours supportée).

Objectifs et approche

Cette thèse adopte une approche pragmatique. Notre objectif est de valider une utilisation étendue de la configuration aux niveaux dénotationnel et opérationnel:

- Du point de vue dénotationnel, nous poursuivons l'effort sur un formalisme commun pour la représentation et la résolution de problèmes de configuration. Nous adoptons l'utilisation du langage relationnel Z comme formalisme de modélisation des modèles objets contraints (MOC). Couplé à une sémantique d'interprétation, la procédure de résolution associée est une recherche énumérative de modèle finis pouvant profiter des avancées de la PC. Nous proposons ensuite l'utilisation de la configuration dans un challenge moderne d'IA : la composition de Services Web Sémantiques (SWS). Cette application illustre l'efficacité de la configuration dans un problème nécessitant un raisonnement complexe et symbolique.
- Du point de vue opérationnel, nous proposons des algorithmes afin de traiter l'explosion combinatoire de la recherche énumérative. Pour les algorithmes complets, nous présentons une méthode d'élimination des isomorphismes qui permet la détection de symétries pour les problèmes de configuration. Nous proposons également une méthode de recherche stochastique basée sur le comportement des colonies de fourmis. Etant donné que les méthodes incomplètes sont un domaine de recherche important pour les CSPs, nous décrivons comment une de ces méthodes peut être appliquée à la configuration.

Contributions originales

Composition de services web sémantiques Notre première contribution est une utilisation industrielle des modèles objets contraints et de la configuration dans un problème d'IA. Le travail présenté n'est pas restreint à une illustration du potentiel de la configuration: nous proposons une description complète et reproductible d'un cadre théorique et expérimental pour la composition de SWS.
Les SWSs sont des agents qui publient leur fonctionnalités et leur comportement de façon à permettre leur découverte, invocation ou composition. Les SWSs communiquent grâce à l'échange de messages. Afin de palier aux incompréhensions, ils reposent sur des conceptualisations partagées des domaines d'application appelées ontologies. La compatibilité entre plusieurs services doit être établie à la fois au niveau abstrait de leurs fonctionnalités et au niveau concret de leur protocole d'interaction. La composition de SWSs requiert la description d'un service composite, dans lequel les services consommés interagissent correctement afin de fournir un service demandé par un utilisateur.
Nous décrivons un modèle objet contraint capable de capturer le comportement des SWSs représentés sous forme de "workflows". Le modèle permet la composition automatique de workflows composites aux niveaux syntaxique et opérationnel. Nous décrivons un deuxième MOC au niveau des fonctionnalités qui peut être utilisé pour compléter

automatiquement des requêtes de compositions. Une traduction de ces requêtes vers le modèle des workflows est proposée afin de prendre en compte ces restrictions abstraites en tant que contraintes additionnelles sur les workflows.
Nous proposons également une méthode pour l'extraction automatique des descriptions du SWS configuré.
Le composeur est validé par son implémentation dans une architecture complète de SWSs développée par le projet européen DIP[1]. En particulier, une description tri-niveau du comportement des SWSs a été co-développée et permet au composeur d'interagir avec les formalismes des autres outils. Les tâches de configurations sont résolues par l'outil JConfigurator de la société ILOG.
Enfin nous présentons des résultats expérimentaux sur des scénarios concrets réalisés dans l'architecture logicielle du projet DIP.

Elimination d'isomorphismes Notre seconde contribution est une méthode d'élimination des isomorphismes qui peut être utilisée pour réduire le nombre de symétries dans les modèles finis en configuration.
Nous proposons un algorithme de complexité en temps pseudo-linéaire (dans le nombre de composants) qui est une approximation de la détection de la canonicité pour les graphes orientés sans circuits (DAGs) colorés (problème de complexité graph-iso complet). La procédure d'énumération maintient une propriété cruciale de rétractabilité canonique permettant de "backtracker" lorsque des configurations isomorphes sont générées durant la recherche. Les résultats théoriques sont appuyés par une série d'expérimentations.

Optimisation par colonies de fourmis Notre troisième contribution est une méthode de recherche stochastique pour la configuration. A partir de recherches existantes sur des algorithmes d'optimisation inspirés du comportement des colonies de fourmis (ACO), nous décrivons une procédure qui traite des problèmes originaux induits par les problèmes de configuration.
Nous proposons une extension du modèle de phéromones capable de représenter des variables ensemblistes et des ensembles non bornés. Nous présentons ensuite des algorithmes basés sur les ACO pour la recherche de modèles finis dans le cadre général des modèles objets contraints.
Enfin, nous fournissons une série de résultats expérimentaux sur des problèmes aléatoires et des benchmarks de configuration. Pour cela, nous utilisons des techniques d'optimisation par essaim de particules afin d'explorer les nombreux paramètres de notre outil.

[1] http://dip.semanticweb.org

2 La configuration en tant que recherche de modèles finis pour les modèles objets contraints

Dans une récente introduction à la configuration, [57] en donne la définition suivante : "configurer est la tâche consistant à composer un système à partir de composants génériques". Le nombre a-priori inconnu de composants nécessaires et le recours nécessaire à la logique des prédicats place la configuration dans le raisonnement du premier-ordre.

2.1 Introduction à la configuration

Modèle de configuration

L'ensemble des systèmes valides est décrit par un modèle de configuration. Celui-ci est souvent représenté à travers les caractéristiques fonctionnelles et techniques des composants potentiels: leur *type*, *attributs* et *relations* avec les autres composants. Le type, combiné à l'héritage, permet de regrouper des caractéristiques communes à plusieurs composants. On obtient alors une *taxonomie* de types. Les attributs représentent des caractéristiques dont le domaine de valeur est fini et d'un type primitif: entiers, flottants, chaînes de caractères, etc. Enfin les relations vers les autres composants peuvent être définis grâce à des *ports de cardinalité*. On peut distinguer les relations de composition, qui expriment l'appartenance de la cible, des relations d'association. L'ensemble des relations est souvent appelé *partonomie*.
La partonomie et taxonomie d'un modèle de configuration peut être représentée grâce à un diagramme de classe UML. La figure 2.1 (page 35) décrit un modèle de configuration pour un ordinateur.
La description d'un système complexe requiert généralement l'expression d'exigences supplémentaires, comme la restriction de l'attribut d'un composant en fonction des composants qui lui sont connectés. Bien que divers formalismes puissent être utilisés, l'utilisation de *contraintes* est la solution la plus répandue grâce à leur puissance expressive et leur modélisation déclarative.
Une contrainte est un prédicat sur des variables. En configuration les variables représentent les types, attributs et relations des composants. Puisque le nombre de composants peut-être indéterminé pendant la modélisation, il faut avoir recours à des *quantificateurs universels*. On peut alors exprimer l'exigence suivante: "Pour tout composant de type t pour lequel 3 composants de type t' sont connectés par la relation r, la valeur de l'attribut a est inférieure à 10".

Problème de configuration

Un problème de configuration est complété par la donnée d'une requête. La requête est un ensemble d'exigences supplémentaires et éventuellement de préférences sur le système recherché. L'objectif est alors de construire et d'exhiber une solution (un modèle), qui est un système valide satisfaisant les exigences de la requête. Nous donnons la définition suivante:

Définition 1 (Problème de configuration) *Un problème de configuration est constitué:*

- *d'un modèle de configuration CM ,*
- *d'une requête R contenant des exigences et/ou des préférences.*

Un configurateur ou procédure de résolution *doit (1) produire un ou plusieurs modèles de CM ∧ R si un tel modèle existe (2) détecter les inconsistances et éventuellement fournir des justifications si un tel modèle n'existe pas.*

Domaines d'application et challenges

La puissance expressive des modèles de configuration permet de représenter de nombreux problèmes. Nous pouvons distinguer deux catégories: les problèmes statiques décrivent des configurations dans lesquelles le nombre de composants est fixé. On peut généralement les modéliser de sorte à fixer le nombre de variables, permettant ainsi l'utilisation de techniques statiques: SAT, CSP, etc. Les problèmes dynamiques, quant à eux, impliquent un nombre indéterminé et parfois non borné de composants.
La configuration est utilisée par l'industrie pour sa capacité à résoudre des problèmes concrets telles que la manufacture ou l'ingénierie (équipements de télécommunications , voitures, ascenseurs, etc.), ainsi que la vente (applications entreprise-client ou entreprise-entreprise). Dans ces applications, l'ensemble des composants disponibles est souvent composé d'objets concrets et (pré)défini dans un "catalogue produit". Seules quelques tentatives ont été effectuées dans des problèmes "académiques" necessistant du raisonnement symbolique et de la création dynamique de composants.
De plus, le contexte général de la configuration pose de nombreux challenges à la communauté scientifique, parmi lesquels la création dynamique de variables et de contraintes, la quantification universelle sur des variables ensemblistes au domaine dynamique, les heuristiques portant sur des décisions de nature différentes, les préférences, les justifications et la mise à l'échelle.

Approches, Formalismes et Méthodes de Résolution

La grande variété dans la structure des solutions, dépendante du domaine d'application, explique le nombre important de techniques appliquées à la configuration. [91, 47, 103] sont des études de différents formalismes existants. Elles classifient les approches par raisonnement à base de règles [67], à base d'étude de cas et à base de modèles [75]. Ces dernières couvrent les logiques de description [76, 68], la programmation par contraintes [74, 33] et les modèles de ressources [56]. Nous pouvons ajouter les modèles à base de connaissances [103], la programmation logique [97], les approches orientées objet [65, 103], SAT [94], les diagrammes de décisions binaires [55], et la sémantique des modèles stables [93].
Les contributions de ces recherches résident soit dans la représentation des modèles de configuration soit dans la procédure de recherche. La plupart de ces méthodes nécessitent

une traduction des modèles de configuration depuis un langage de haut niveau vers le formalisme choisi.
Dans cette thèse nous retenons une approche orientée objet [65]. Celle-ci combine la programmation par contraintes et la logique des descriptions. Le langage résultant profite de la logique des prédicats apportée par les contraintes, et ses termes sont soit des variables logiques soit des symboles de la logique des descriptions. La procédure de recherche, quant à elle, profite de l'efficacité des méthodes de résolution issues de la programmation par contraintes.
Nous présentons une formalisation et une généralisation de cette représentation sous la forme de *modèles objets contraints* associés à une sémantique des modèles finis et à une méthode de recherche énumérative.

2.2 Formalisation des modèles objets contraints

Nous avons choisi de formaliser nos modèles objets contraints à l'aide du langage relationnel Z. Ce choix est motivé par les raisons suivantes:

- Z offre un langage commun pour la déclaration de classes, relations et contraintes,
- Z est au moins aussi expressif que la logique des prédicats,
- ses fondations mathématiques permettent les preuves et la reproduction des résultats,
- le modèle déclaratif est indépendant de la méthode de résolution,
- nous poursuivons l'effort, initié par Laurent Henocque, visant à proposer un langage commun pour la communauté.

Voici un exemple de définitions Z, correspondant à un sous-modèle de la Figure 2.1 (page 35), représenté sur la figure 2.9 (page 54):

$$
\begin{array}{l}
\forall\, mb : Motherboard\ ;\ r : Ram \mid r \in mb.ram \bullet mb.typeRam = r.type \\
\forall\, mb : Motherboard\ ;\ r : Ram\ ;\ p : Processor \\
\quad \mid r \in mb.ram \land p \in mb.processor \\
\quad \bullet\ mb.totalPrice = \\
\qquad mb.price + bagsum((mb.ram) \rightsquigarrow price) + bagsum((mb.processor) \rightsquigarrow price) \\
\forall\, mb : Motherboard \mid \#(mb.processor) = 2 \bullet \\
\quad \forall\, p_1, p_2 : Processor \mid p_1 \in mb.processor \land p_2 \in mb.processor \bullet \\
\qquad p_1.speed = p_2.speed
\end{array}
$$

2.3 Sémantique des modèles finis

Nous définissons la sémantique des modèles objets contraints à travers la spécification d'une fonction d'interprétation.

Définition 2 (Interprétation d'un modèle objet contraint) *L'interprétation $I(M)$ d'un modèle objet contraint M est définie par la généralisation récursive de l'interprétation de ses éléments. Les éléments classes, objects, attributes et relations ont leur propre interprétation; tandis que les contraintes ont l'interprétation usuelle de la logique des prédicats.*

Définition 3 (Modèle fini) *Un modèle fini est un ensemble fini d'objets satisfaisant l'ensemble des contraintes définies sur ces objets.*

2.4 Méthodes de recherche de modèles finis pour les modèles objets contraints

Dans cette thèse nous retenons les algorithmes basés sur une recherche énumérative des modèles finis, ayant notamment prouvé leur efficacité dans le domaine de la programmation par contraintes.

Définition 4 (Espace de configuration) *L'espace de configuration est l'ensemble des combinaisons potentielles d'un modèle de configuration (ou modèle objet contraint).*

Si l'espace de configuration est fini, une recherche énumérative peut être effectuée en un temps fini. Bien que ce ne soit pas le cas général de la configuration, plusieurs restrictions permettent de garantir cette propriété.

Amélioration des méthodes de recherche A l'instar des CSPs, les méthodes de recherches énumératives peuvent bénéficier de nombreuses techniques d'amélioration : heuristiques de choix de variable et de valeurs, réduction des domaines par propagation des contraintes, détection des symétries, etc. Cependant nous avons déjà souligné que leur application à la configuration soulève un certain nombre de problèmes originaux. Nous traitons la question des symétries dans le chapitre 5. La recherche d'un modèle, tout comme pour les CSPs, peut se faire du point de vue de la satisfaction (trouver un modèle) ou de l'optimisation (trouver le "meilleur" modèle pour un critère donné).
Les algorithmes incomplets, qui explorent partiellement l'espace de recherche, peuvent également être appliqués à la configuration. Parmi ceux-ci, les algorithmes stochastiques combinent heuristiques et choix aléatoires. Nous proposons un algorithme stochastique pour la configuration dans le chapitre 6.

JConfigurator JConfigurator est un configurateur java développé par la société ILOG. L'outil utilise une méthode de recherche complète énumérative, basée sur la méthode des tableaux généralisée [58]. Nous utilisons JConfigurator pour l'implémentation de notre composeur de services web sémantiques dans le chapitre 4.

3 Introduction à la composition de services web sémantiques

3.1 Introduction aux services web sémantiques

Définition 5 (Services Web Sémantiques) *Les SWSs sont des agents logiciels qui publient leurs caractéristiques fonctionnelles et comportementales de façon à permettre leur découverte, invocation et composition à travers du raisonnement automatique.*

L'interaction et le raisonnement sont possibles grâce à des conceptualisations partagées appelées ontologies. On distingue deux types d'ontologies. Les ontologies de données décrivent la connaissance d'un domaine, et sont utilisées dans les messages échangés par des SWSs. Les ontologies de services décrivent quant à elles ce que le service est capable d'effectuer ainsi que la manière d'interagir avec le service.

Ontologies de données

Définition 6 (Ontologie de donnée) *Une ontologie de données est un modèle de connaissances d'un domaine représentant des concepts à travers les notions de classes, attributs et relations.*

Deux principaux modèles d'ontologies de données ont émergé ces dernières années: OWL[2] (Ontology Web Language) et WSML[3] (Web Service Modelling Language). La figure 3.2 (page 65) présente un fragment d'ontologie des vins et sa description en OWL et WSML.

Ontologies de services

A l'instar des ontologies de données, les principales ontologies de services utilisées sont OWL-S et WSMO. La description des caractéristiques fonctionnelles est proche dans les deux approches. Le *service profile* de OWL-S et la *capabilité* de WSMO fournissent un ensemble de messages d'entrée (inputs) et de sortie (outputs), dont le type est pris dans une ontologie de données, et qui représente respectivement ce que le service requiert et ce qu'il peut fournir. La description est parfois complétée par un ensemble de préconditions et d'effets.
En ce qui concerne les caractéristiques comportementales, les deux approches diffèrent à la fois dans les informations représentées et dans le formalisme utilisé. Nous retiendrons le double point de vue, plus précis, apporté par WSMO. Dans celui-ci le service contient une choréographie, qui décrit de quelle façon un client peut interagir avec le service, et l'orchestration, qui décrit comment le service réalise en interne ses fonctionnalités, notamment grâce à l'utilisation de SWSs externes.

Définition 7 (choréographie) *Une choréographie est une description comportementale d'un service permettant à un client d'interagir correctement pour réaliser les fonctionnalités du service.*

[2] http://www.w3.org/Submission/OWL-S/
[3] http://www.wsmo.org/

Définition 8 (Orchestration) *Une orchestration est une description comportementale d'un service indiquant comment un service composite utilise des SWSs externes afin de réaliser ses fonctionnalités.*

La figure 3.4 (page 67) donne un aperçu d'une telle ontologie de services. La description du comportement est généralement effectuée via la notion de processus métier (ou "workflow"). Il existe de nombreux formalismes pour spécifier des workflows, reposant souvent sur les réseaux de Petri (diagrammes d'activité UML2, YAWL) ou sur l'algèbre des processus (Pi-calcul).

Goals et discovery

La recherche et la découverte d'un SWS repose sur la notion de *goal*. Un Goal est un ensemble de fonctionnalités requises, qui peuvent s'exprimer selon le même formalisme qu'une capabilité. Un goal est parfois doublé d'exigences comportementales.
Le processus de *discovery* est la tâche consistant à déterminer quel ensemble de services peuvent être utilisés pour un goal donné.

3.2 Introduction à la composition

La composition peut être définie comme "la tâche consistant à combiner et coordonner un ensemble de SWS afin d'obtenir une fonctionnalité". Elle peut s'établir au niveau abstrait des goals, et requiert alors une deuxième phase où des services concrets seront sélectionnés et incorporés dans une orchestration. Elle peut également s'établir au niveau concret des processus, produisant un workflow complet sur la base de services existants.

Définition 9 (Problème de composition de SWS niveau goal) *Soit R une requête, G une librairie de goals (ou capabilités) disponibles, Onto un ensemble d'ontologies. Un composeur niveau goal produit une description abstraite d'un service web composite qui peut réaliser les fonctionnalités requises dans R. Cette description est constituée d'un ensemble de goals $g \in G$ nécessaires et d'une série d'exigences sur l'orchestration potentielle.*

Définition 10 (Problème de composition de SWS niveau processus) *Soit R une requête, C une librairie de choréographies disponibles, Onto un ensemble d'ontologies. Un composeur niveau processus produit la description d'un service web composite qui peut réaliser les fonctionnalités requises dans R. Cette description est une orchestration O.*

3.3 Composition de SWS: approches pratiques

Il existe de nombreux travaux dans le domaine de la composition de SWS. Une grande partie introduit des langages et formalismes pour la composition manuelle, la vérification ou le diagnostique. Nous nous intéressons dans cette thèse aux approches scientifiques traitant de la composition (semi-)automatique. Des formalismes de nature très différente

ont été utilisés, et le niveau de fonctionnalités atteint est également variable. Le tableau 3.1 (page 76) donne un comparatif des approches connues en fonction de critères exprimant le niveau de composition atteint.

4 Composition de services web sémantiques à base de configuration

4.1 Une approche à deux niveaux à base de configuration

Nous proposons dans cette thèse une application de la configuration à deux niveaux: un langage de requêtes de composition configurable est utilisé pour guider la configuration des processus.

Composition au niveau processus

Une composeur niveau processus doit produire une orchestration à partir de choréographies de SWS disponibles. Nous considérons que les orchestrations et les choréographies sont décrites via des workflows. Nous définissons pour cela un langage de workflows (AD-S), qui est un sous-ensemble des diagrammes d'activité UML2, basé sur la sémantique des réseaux de Petri.

Configuration de workflows Nous définissons un MOC capable de capturer les workflows exprimés avec ce langage AD-S. La requête de configuration est un ensemble de messages de sortie (outputs) requis par l'utilisateur et un ensemble de messages d'entrée (inputs) qu'il est capable de fournir. Le configurateur crée alors un workflow composite réalisant ces fonctionnalités, contenant les choréographies de SWS utilisées et des éléments internes.
Le workflow crée est valide du point de vue syntaxique grâce aux contraintes posées dans le MOC. D'un côté, le workflow respecte les restrictions du langage. De l'autre, tous les messages d'entrée du workflow sont connectés à un message de sortie compatible par rapport aux ontologies de données concernées.
Le workflow configuré garantit également qu'au moins un de ses chemins d'exécution potentiels mène aux objectifs de l'utilisateur. Pour cela, un attribut nomme "active' est ajouté aux noeuds et arcs du workflow. Les contraintes posées sur cet attribut simulent la sémantique d'exécution du workflow de telle façon qu'un noeud "actif" possède au moins un chemin d'exécution potentiel.

Extraction des descriptions du SWS composite La description complète d'un SWS composite nécessite de fournir sa capabilité, choréographie et orchestration. L'obtention de ces informations n'est pas trivialement réalisable car elle nécessite de transposer la logique des choréographies des SWS composées vers l'orchestration, puis vers la choréographie du SWS composite. Nous proposons une procédure post-configuration permettant de traiter ce problème.

L'ensemble de la procédure de composition niveau processus peut être visualisé sur la Figure 4.1 (page 81).

Composition au niveau goal

Nous définissons au niveau abstrait des goals un langage de requêtes de composition, appelées goals de composition, permettant de spécifier un ensemble d'exigences sur l'orchestration. Ces exigences sont prises en compte lors de la configuration de workflows grâce à une traduction modulaire des goals de composition vers des contraintes et éléments du MOC des workflows.

Interaction avec la discovery Les goals de composition permettent de spécifier les goals "atomiques" qui sont nécessaires pour une fonctionnalité donnée. Via la discovery, on obtient alors une librairie de SWS utiles. Ce processus réduit significativement le nombre de choréographies candidates lors de la composition de workflows.

Configuration des requêtes Une autre originalité de notre approche est le fait que les goals de composition soient eux-mêmes configurable, car nous les décrivons de nouveau à l'aide d'un MOC. Cela permet une complétion automatique ou semi-automatique des requêtes. Celles-ci étant définies au niveau abstrait des fonctionnalités, nous obtenons une composition au niveau goal.
L'ensemble de la procédure de composition double niveau peut être visualisé sur la Figure 4.2 (page 82). Un exemple sur un scénario concret est présenté sur la figure 4.3 (page 83, une requête de composition), la figure 4.4 (page 83, un goal de composition configuré), et la figure 4.5 (page 84, l'orchestration calculée). Ce scénario, développé conjointement avec un partenaire industriel du projet européen DIP (British Telecom), requiert la composition d'un service capable de proposer un modem, un PC et une connexion internet. On peut noter que dans le goal de composition configuré un ensemble de goals atomiques ont été sélectionnés. Dans l'orchestration générée, on retrouve un ensemble correspondant de SWS composés.

4.2 Implémentation du composeur, intégration dans le projet DIP et expérimentations

Intégration dans le projet DIP

Le projet est essentiellement basé sur l'architecture WSML (langage basé sur les machines d'états abstraites ASM), WSMO (ontologies) et WSMX (moteur d'exécution). Le moteur IRS-III (Internet Reasoning Service), qui offre un support direct de WSMO, est également disponible. Cette architecture permet d'utiliser différents outils du domaine des SWS : discovery, registres, médiation, modélisation.
L'interaction avec ces outils est obtenue grâce à l'intégration de notre langage AD-S dans une description tri-niveaux des orchestrations et des choréographies: une ontologie des ASM, une évolution du langage de calcul des processus Cashew-S (utilisé par IRS et

supportant WSMO), ainsi que notre modèle AD-S. La figure 4.25 (page 125) montre les caractéristiques de cette description tri-niveaux. La grammaire WSML a été étendue afin de pouvoir exprimer chacun de ces langages, et des traductions ont été développées.

Implémentation

Le composeur a été implémenté avec le langage JAVA. Les tâches de configuration sont réalisées par le configurateur JConfigurator de la société ILOG, grâce à une traduction de nos MOCs vers le langage de l'outil. Aucune heuristique n'a été utilisée en dehors des heuristiques par défaut de l'outil.

Résultats expérimentaux

Nous avons conduit des expérimentations sur quatre scénarios. Ceux-ci ont été intégrés dans une architecture complète de SWS, depuis la composition jusqu'à l'exécution. Les résultats sont donnés sur le tableau 4.3 (page 127).
Deux de ces scénarios (PS-a et PSBank) n'utilisent pas les goals de composition. Ils peuvent servir de comparaison avec des approches existantes. Le scénario PS-b est une version modifiée proposée par un partenaire industriel du projet (SAP), dans laquelle les choréographies ont été largement complexifiées. Enfin le scénario NMPC-bundle a déjà été présenté dans les sections précédentes.
Les résultats sur les scénarios PS-a et PSBank montrent l'efficacité du composeur quand le nombre de services disponibles est restreint. Cependant on note aussi les problèmes dus à l'explosion combinatoire quand ce nombre de services est augmenté. L'impact des goals de composition dans les scénarios PS-b et NMPC-bundle est évident: grâce à l'interaction avec la discovery le nombre de SWS candidats est réduit et minimise donc l'espace de recherche du configurateur. Cependant le problème d'explosion combinatoire survient toujours si plusieurs services offrent les mêmes fonctionnalités. Un autre apport des goals de composition est visible pour le scénario NMC-bundle: grâce à la propagation des exigences (comme le type de connexion), nous diminuons de nouveau le nombre de SWS candidats. De plus, les contraintes supplémentaires apportées par un goal de composition réduisent l'espace de recherche par le rejet d'orchestrations au comportement non désiré.

Comparaison aux approches existantes

Du point de vue opérationnel, le tableau 4.5 (page 129) compare nos résultats à l'approche ayant les meilleures performance à notre connaissance. On peut voir que notre approche est clairement compétitive.
Du point de vue des fonctionnalités, le tableau 4.4 (page 128) présente le niveau de composition atteint par notre outil en fonction des critères utilisés pour comparer les approches existantes. Encore une fois notre méthode à base de configuration est largement comparable et souvent plus complète. Les limitations se trouvent dans le raisonnement sur les ontologies encore restreint, et dans le support de certains "workflows patterns" comme les boucles.

5 Rejet des isomorphismes en configuration

Nous proposons une méthode permettant d'améliorer l'efficacité des recherches énumératives de modèles finis. Une difficulté inhérente à la résolution de problèmes de configuration est l'existence de nombreuses structures isomorphes. Ces structures mènent à des solutions équivalentes.
Nous poursuivons ici des travaux sur la génération d'arbres canoniques pour des problèmes de configuration impliquant uniquement des relations de composition [44, 51]. Nous généralisons la méthode aux graphes orientés sans circuits (DAGs), traitant ainsi l'ensemble des structures de configuration.

5.1 Génération d'arbres canoniques en configuration

Les travaux présentés dans [44, 51] décrivent une procédure générant uniquement des arbres canoniques. La canonicité est définie comme le représentant minimal d'après un ordre total établi sur les arbres, appelés *T-trees*. Une procédure de génération d'arbres canoniques, de complexité en temps $O(n \log n)$, est alors définie. Celle-ci est prouvée complète, non redondante, et permet de "backtracker" sur les arbres non canoniques.

5.2 Généralisation au traitement des structures isomorphes en configuration

Génération de DAGs faiblement canoniques

La détection de DAGs canoniques est un problème ouvert classé *graph-iso complet*, dont la complexité est au moins NP. Nous proposons une procédure permettant de générer uniquement des DAGs appelés faiblement canoniques, définis par le fait que leur arbre couvrant minimal est un T-tree canonique. La procédure, dont l'algorithme de rejet reste de complexité pseudo-linéaire, est prouvée complète, non redondante et conserve la propriété de pouvoir "backtracker" sur des graphes isomorphes. La procédure est basée sur une complétion des arbres canoniques pendant laquelle seules des arcs sont ajoutés. Elle est présentée sur le tableau 5.1 (145).

Exploitation des symétries

La méthode est améliorée par l'exploitation des symétries détectées sur les arbres canoniques. Bien que la détection de symétries soit un problème difficile dans le cadre général des graphes, elle est directe sur les arbres. Les symétries ont une conséquence évidente sur les isomorphismes: si deux noeuds sont symétriques, alors l'ajout d'une arête à l'un ou à l'autre de ces noeuds produit deux graphes isomorphes.

5.3 Expérimentations

Des résultats expérimentaux sont présentés sur le tableau 5.2 (148). On constate une nette diminution dans le nombre de graphes générés, ainsi que la complémentarité des deux techniques. Ceci laisse entrevoir la possibilité d'augmenter significativement la

taille des problèmes de configuration pouvant être traités.
Nous avons également combiné l'utilisation de notre procédure avec Nauty, un outil permettant de détecter de manière complète (mais de complexité linéaire NP-complet) les symétries dans un graphe. On constate que de nombreux isomorphismes subsistent dans les graphes générés grâce à notre procédure, mais que leur détection avec Nauty devient rapidement coûteuse. Ces résultats laissent à penser que l'ajout de nouvelles techniques peut permettre de détecter davantage d'isomorphismes, tout en conservant une complexité pseudo-linéaire.
Une limitation importante de ce travail est qu'il n'a pas été intégré dans une procédure complète de configuration. A ce niveau, les symétries détectées durant la génération des structures peuvent servir à réduire la complexité du problème d'instanciation associé. En effet le problème, ayant alors un nombre fixé de variables, peut être considéré comme un CSP "classique" et permet l'exploitation déjà largement étudiée des symétries.

6 Recherche stochastique en configuration: colonies de fourmis

Une des difficultés inhérente à la recherche énumérative est l'explosion combinatoire du nombre de solutions potentielles. Cela est d'autant plus visible dans les problèmes d'optimisation, où l'on ne recherche pas simplement une solution mais la meilleure selon une fonction d'évaluation donnée. De nombreux travaux cherchent à limiter les effets de cette explosion dans les méthodes complètes : filtrage, symétries, décomposition, heuristiques, etc. D'autres travaux utilisent des recherches *incomplètes* qui n'explorent que partiellement l'espace de recherche. En particulier, les méthodes dites *stochastiques* combinent des méthodes aléatoires et heuristiques afin de trouver rapidement une solution.
Nous proposons l'application à la configuration d'algorithmes basés sur le comportement des colonies de fourmis. Ceux-ci, basés sur la méta-heuristique ACO (Ant Colony Optimization), ont déjà prouvé leur efficacité dans le domaine des CSPs.

6.1 Introduction à la méta-heuristique ACO

La méta-heuristique ACO tire parti d'un processus auto-catalytique distribué, observé dans les colonies de fourmis, basé sur les *phéromones*. Une phéromone est une substance chimique déposée sur un chemin jugé prometteur par une fourmi (par exemple vers une source de nourriture). Les autres fourmis sont influencées dans leur choix par les phéromones précédemment déposées.
Dans un problème d'optimisation, une phéromone est une valeur déposée par une fourmi artificielle sur les éléments d'une instanciation partielle jugée interessante. Un modèle de phéromones est ainsi crée et complété successivement par des fourmis artificielles, qui sert d'heuristique à l'algorithme de recherche de solutions. Un algorithme générique basé sur ACO est présenté dans le tableau 6.1 (page 155). Un algorithme ACO est sujet à de nombreux paramètres. Un paramètre essentiel est l'*évaporation* qui définit une

diminution de l'ensemble des phéromones du modèle à chaque itération, préalablement au dépôt, permettant ainsi de diversifier la recherche par un "oubli" progressif.

6.2 ACO pour la configuration

En configuration, l'aspect dynamique des méthodes de recherche (génération de composants, ensemble non borné de variables) pose des problèmes originaux dans l'application du cadre ACO. En effet, contrairement au cadre original, une décision peut impliquer le choix d'un certain nombre d'éléments parmi un ensemble non borné. Par exemple, le choix des composants cibles d'une relation.

Modèle de phéromones

Nous proposons une méthode permettant de simuler un ensemble non borné à travers un ensemble borné évolutif. L'idée principale est qu'un tel ensemble est scindé en deux catégories par un séparateur. Quand une valeur appartenant à la deuxième catégorie est choisie, l'ensemble est augmenté et le séparateur modifié. Une des propriétés est que l'ensemble des valeurs possibles peut être ajouté au cours des itérations de l'algorithme. Nous définissons ensuite un modèle de phéromones adapté aux décisions effectuées par un configurateur.

Algorithmes

Nous proposons deux algorithmes exploitant ce modèle de phéromones. Ils sont présentés dans les tableaux 6.4 (page 164) et 6.5 (page 165). Dans le premier, le nombre de composants est choisi en premier lieu, puis les composants sont instanciés. Dans le second, les composants sont générés durant le parcours en profondeur d'un *graphe de construction*.

6.3 Implémentation et Expérimentations

Les algorithmes sont implémentés en JAVA, et s'appuient sur une librairie des modèles objets contraints développée en parallèle. Aucune heuristique (en dehors des phéromones) n'a été utilisée dans les expérimentations présentées.

Paramètres et optimisation par essaim de particules

L'outil dispose d'un grand nombre de paramètres. Afin d'optimiser leurs valeurs, nous avons utilisé le concept d'*optimisation par essaim de particules* (PSO). Dans ce cadre, une particule, qui représente l'exécution de l'algorithme avec un jeu de paramètres fixé, évolue dans un espace de valeurs (pour les paramètres). Le mouvement d'une particule à chaque itération est influencé par une technique aléatoire, par son meilleur jeu de paramètres jusqu'à présent, et par celui de la meilleure particule. La méthode permet de converger vers des paramètres optimaux.

Expérimentations

Diverses expérimentations ont été réalisées pour des problèmes aléatoires et pour des "benchmarks" connus en configuration. Les résultats sont présentés dans les tableaux 6.7 (page 169) et 6.8 (page 170), ainsi que sur les figures 6.3 (page 172) et 6.4 (page 173).
Les résultats offrent une validation partielle de l'approche. Sur des problèmes simples, les algorithmes sont efficaces et montrent la convergence des algorithmes. Sur des problèmes plus complexes, les solutions optimales ne sont pas régulièrement trouvées mais des solutions de qualité moindre sont tout de même obtenues. Les expérimentations qui se concentrent sur la taille des solutions montrent également le bon comportement des algorithmes vis à vis des contraintes du problème, validant ainsi le modèle phéromonal. Toutefois nous soulignons le caractère préliminaire de ces travaux, et la nécessité à la fois d'étendre le nombre d'expérimentations et d'améliorer les algorithmes présentés avec des heuristiques et des techniques éprouvées en recherche incomplète.

7 Conclusion et perspectives

Dans cette thèse nous avons utilisé la recherche de modèles finis dans le contexte de modèles objets contraints exprimés grâce au langage Z. La combinaison de la programmation par contraintes, des logiques de description et des théories du premier ordre permet de représenter de nombreux problèmes d'IA de manière totalement déclarative. De plus, le formalisme présenté est indépendant de la procédure de recherche. Puisque la résolution s'applique directement au niveau du langage déclaratif, il n'est pas nécessaire de recourir à des traductions et les résultats peuvent être exploités directement.
Nous avons présenté une application à la composition de SWS qui illustre ces avantages. Le cadre proposé agit à la fois au niveau abstrait des fonctionnalités et au niveau concret des workflows. A chaque niveau, les ontologies de données font partie du modèle, permettant le raisonnement sans recours à des formalismes additionnels. Dans le modèles des workflows, nous combinons des propriétés syntaxiques et opérationnelles. Dans le modèle des goals de composition, un point de vue original est adopté à travers la configuration de requêtes de composition. Nous avons également soulevé des questions originales comme l'extraction automatique de l'orchestration et de la choréographie du service composite.
L'application est validée par son intégration dans une architecture complète de SWS. Les orchestrations générées peuvent notamment être directement utilisées dans un moteur d'éxécution. Comparé aux approches existantes, notre composeur est compétitif à la fois dans les fonctionnalités offertes et dans les résultats opérationnels. Cependant, l'approche souffre de difficultés de mise à l'échelle liées à la fois au contexte particulier du web et à la méthode de résolution.
En effet, la nature énumérative de la recherche de modèles finis induit une explosion combinatoire bien connue dans le domaine des CSPs. De nombreuses techniques ont été développées pour augmenter la taille des problèmes traités par les CSPs. Nous avons exploré l'application de méthodes similaires à la configuration, qui prennent en compte

ses propriétés particulières.
Pour la recherche exhaustive de modèles finis, nous avons décrit une méthode d'élimination d'isomorphismes. Nous avons généralisé aux DAGs un travail existant sur les arbres, traitant ainsi la symmétrie des structures de configuration générées dynamiquement. La procédure présentée s'appuie sur un algorithme pseudo-linéaire permettant de détecter des structures isomorphes, tout en maintenant une procédure cruciale de rétractabilité canonique. Les résultats théoriques et expérimentaux sont prometteurs mais n'ont pas été appliqués à des problèmes concrets de configuration.
Enfin nous avons décrit une méthode stochastique pour la recherche de modèles finis. A notre connaissance, ce domaine de recherche intense en programmation par contraintes n'avait pas encore été appliqué aux théories du premier ordre. L'approche présentée se place dans le cadre de recherches existantes sur le comportement des colonies de fourmis appliqué aux algorithmes d'optimisation, et traite des nombreux problèmes soulevés par la configuration. Bien que les résultats obtenus soient prometteurs, le domaine offre clairement de nombreuses pistes d'amélioration.
Nous pensons avoir contribué à une utilisation générique de la configuration en IA. Nous avons en effet généralisé la recherche de modèles finis dans un contexte combinant l'efficacité opérationnelle à des structures dynamiques et du raisonnement symbolique.

Perspectives

La recherche de modèles finis pour les modèles objets contraints est un paradigme logique puissant dont les perspectives pratiques et théoriques sont nombreuses.
L'application à la composition de SWS ne couvre pas toutes les fonctionnalités possibles. La modélisation de "workflows patterns" supplémentaires ou de la compensation sont envisageables, mais pourrait révéler des limitations dans la capacité expressive de la configuration. Les garanties d'exécution des workflows configurés peuvent également être étendues en prenant en compte, par exemple, la multiplicité des tokens ou l'exclusion mutuelle de leur domaine de valeurs.
De nombreuses autres applications de la configuration peuvent être explorées. De plus, le résultat du processus de configuration peut lui-même être utilisé comme modèle pour un autre problème, ouvrant des perspectives d'architectures à objectifs multiples. Prenons par exemple l'extraction de la sémantique d'un texte descriptif [28], qui crée un modèle du monde connu. Si l'on considère ce résultat comme une ontologie, il peut être utilisé comme une requête en langage naturel pour le composeur ou tout autre application basée sur la configuration.
Sur le plan théorique, le raisonnement au niveau méta-modèle est une perspective importante car son potentiel expressif est considérable. Cependant cela implique une modification fondamentale de la configuration entraînant des problèmes de modélisation et de résolution.
Comme pour les autres domaines d'IA, l'utilisation pratique de la configuration dépend de son efficacité opérationnelle. De ce point de vue le travail présenté offre de nombreuses perspectives.
En ce qui concerne l'élimination des isomorphismes, d'autres méthodes pseudo-linéaires

pourraient augmenter le nombre de structures équivalentes détectées. Afin d'appliquer l'approche dans des problèmes concrets de configuration, nous prévoyons d'utiliser la librairie java des modèles objets contraints développée pour notre méthode stochastique, et d'utiliser les symétries détectées dans l'instanciation des composants.
Dans l'application des ACO à la configuration, nous envisageons de nombreuses pistes de recherche. Tout d'abord, nous ajouterons des heuristiques de variables et de valeurs pendant la recherche. Nous étendrons ensuite les expérimentations afin d'analyser plus profondément le comportement de l'algorithme et l'améliorer. Nous envisageons notamment de le diversifier grâce à des phases d'exploration et d'intensification.
Enfin, nous désirons poursuivre l'effort d'un formalisme commun de configuration basé sur le langage Z, permettant de comparer différentes méthodes de résolution. Dans ce but, nous prévoyons de définir une traduction modulaire de Z vers notre librairie de modèles objets contraints. La combinaison de la représentation des MOCs en Z et de méthodes de résolution offrira alors un langage complet d'IA avec une sémantique opérationnelle des modèles finis.

Chapter 1

Introduction

This thesis aims at extending the practical use of constraint programming (CP) in artificial intelligence (AI). More precisely we wish to support the claim that configuration, as an enumerative finite model search of constrained object models, is a viable and well-fitted option for AI problems of first-order theories.
Constraint programming has become a major field of research in artificial intelligence, in particular with the notion of Constraint Satisfaction Problems (CSP). CSPs combine a completely declarative model with solving algorithms that have proven efficient in many problems. Indeed, several methods have been developed to deal with the combinatorial explosion inherent to enumerative search: variable and value heuristics, maintaining consistency, domains reduction via constraints propagation, symmetry breaking, stochastic search, etc.
Different extensions of CSPs have been proposed to handle problems of first-order logic. However their practical application raises many issues. From the modelling point of view, difficulties arise when one tries to express constraints in the presence of a dynamic number of (set-)variables. From the solving point of view, first-order theories induce a complexity increase and the use of classical CSPs search enhancement methods introduces a set of original challenges.
Configuration problems involve the creation of complex structures with a dynamic number of interconnected components. CP approaches, through extensions of CSPs, have emerged as a potentially adapted formalism besides the afored mentioned issues. In that respect, object-oriented configurators which combine description logics with constraint programming, offer several advantages. The approach provides a high degree of freedom in the declaration of the model and its constraints together with the efficiency of CP problem-solving methods.
However despite its inherent potential, configuration is seldom considered as an option for general AI problems involving symbolic reasoning. This may be due to its tradition of manufacturing application, to the expressive restrictions imposed by existing configurators, or to the lack of a common formalism for problems representation and solving. Indeed, most existing configurators use their own tool-oriented formalism. It may restrict the general configuration context (for instance most of them do not fully support the

dynamic creation of variables) and is often closely related to a specific solving procedure.

1.1 Objectives and Approach

This thesis adopts a pragmatic standpoint. Our objective is to validate an extended use of configuration at both the denotational and operational level:

- At the denotational level, we follow the effort on a common and general formalism for representing and solving configuration problems. We advocate the use of the relational language Z as a modelling formalism for constrained object models (COM). Together with interpretation semantics, the related solving procedure can be an enumerative finite model search which may benefit from existing work in CP.
  We then propose the use of configuration for a modern AI challenge: the composition of Semantic Web Services (SWS). The application illustrates how complex and symbolic reasoning can be efficiently achieved with configuration.

- At the operational level, we propose algorithms to deal with the inherent combinatorial explosion of enumerative search. For complete algorithms, we present an isomorphism rejection method as a symmetry breaking method for configuration problems.
  We also propose a stochastic search method based on ant colonies behaviour. The use of incomplete methods is an intense field of research in CSPs. We describe how it can be applied to configuration.

1.2 Original Contributions

Composition of semantic web services Our first contribution is an industrial use of constrained object models and configuration on an AI problem: the automatic composition of semantic web services. The presented work is not restricted to an illustration of configuration's potential: we present a complete and reproducible description of a theoretical and experimental framework for SWS composition.
SWSs are software agents which publish their functionalities and behavioural interfaces so as to allow reasoners to help discover, invoke, compose or adapt them. SWSs communicate through the exchange of messages. To account for potential misunderstandings, SWSs rely on shared conceptualizations of domains called ontologies. The compatibility must be established both at an abstract level, with respect to services functionalities and data carried by messages, and at a concrete level, with respect to their communication protocol or message exchange patterns. SWS composition requires to design a composite service where consumed services correctly interact in order to achieve user's

requirements.
We describe a COM able to capture the behaviour of SWSs through a workflow representation. The model allows for the process-level automatic computation of composite workflows from a syntactical and executional standpoint.
We describe another COM at the level of functionalities which can be used to automatically complete composition requests. A translation from these requests to the workflow model is proposed in order to take into account its set of abstract requirements as additional workflow constraints.
We also propose a method for the automatic extraction of the related SWS descriptions from the configured composite workflow.
The composer is validated by its implementation in a full SWS framework developed with the DIP[1] European project. In particular, a 3-level description of SWSs behaviour has been joint developed and allows the composer to interact with the formalisms used by other tools. Configuration's tasks are handled with ILOG's tool JConfigurator.
Finally, we present experimental results on real-world scenarios which have been demoed within the project's architecture.

Isomorphism rejection Our second contribution is an isomorphism rejection method that can be used to break the symmetries existing among finite models.
We propose a pseudo linear time algorithm (in the number of vertices) that approximates vertex colored DAG canonicity detection (a graph-iso-complete problem). The enumeration procedure maintains a crucial canonical retraction property allowing early backtrack when generating isomorphic configurations during the search.
The theoretical results are backed by a range of experiments.

Ant Colonies Optimization for configuration Our third contribution is a stochastic search method for configuration. Based on existing work on optimization algorithms inspired by the behaviour of ant colonies (ACO), we describe a framework which deals with the original issues of general configuration problems.
We propose an extension of the pheromone model able to handle set-variables and unbounded sets. We then present ACO-based algorithms for finite model search in the general context of constrained object models.
We provide a set of experiments on both random problems and known benchmarks for configuration. We use Particle Swarm Optimization to explore the large parameter space of our tool.

1.3 Plan of the thesis

Chapter 2 is a presentation of configuration. We present in Section 2.1 its original challenges and a state of the art of existing tools. In the remaining Sections we describe its generalization to finite model search for constrained object models expressed using the Z language.

[1] http://dip.semanticweb.org

Chapter 3 introduces the context for Semantic Web Services Composition. We present SWSs concepts and formalisms in Section 3.1 while Section 3.2 discusses the particular problem of composition. Section 3.3 is a state of the art of existing approaches.
Chapter 4 describes the configuration-based framework for SWS composition. We first give an overview of our approach in Section 4.1. We then describe in Section 4.2 the AD-S workflow language and COM for process-level composition. Section 4.3 describes our configurable request language and its translation to an extended workflow model. A method for the automatic extraction of the composite SWS descriptions is given in Section 4.4. Our implementation in the project's framework and a set of experiments is provided in the remaining Sections.
Chapter 5 describes our isomorphism rejection method for configuration. We discuss the problem and its relation to classical symmetry breaking in Section 5.1. We then briefly present the existing work on a tree generation procedure in Section 5.2. Our generalization to DAGs is described in Sections 5.3 and 5.4. Experiments are then provided.
Chapter 6 describes our stochastic search procedure. We present the ACO metaheuristic, its application to CSPs, and discuss the original issues brought by configuration in Section 6.1. Section 6.2 describes our pheromones model and algorithms. A set of experiments is provided in Section 6.3.
Finally Chapter 7 is a synthesis of our work where we propose directions for future work.

Chapter 2

From Configuration to Finite Model Search for Constrained Object Models

In a recent introduction to configuration [57], U. Junker defines it as "the task of composing a customized system out of generic components". Based on a modelled knowledge of acceptable systems and a set of problem-specific requirements, the objective is to construct and exhibit a solution (a *model*). The a-priori unknown number of components required in a solution and the need for predicate logic places configuration in the scope of first-order logic reasoning tasks.
In this chapter we first present configuration and a range of covered problems. We point out the main challenges in modelling and solving and discuss how configuration relates to other artificial intelligence fields. We then survey the formalisms and search methods that have been developed through years, and show how constrained object models (COMs) are well-suited to represent a general configuration problem. Finally we present a detailed formalism for COMs, based on the Z relational language, which we will use throughout the document. We also describe its finite model semantics.

2.1 Introduction to configuration

2.1.1 A history of configuration

First steps

The term configuration first appeared in the 80's during researches on rule-based expert systems [67] for assembling a computer system out of predefined and connected components. Configuration then evolved to cover several system design tasks like networks, circuit boards or buildings. Frayman and Mittal gave a first definition of configuration in [75], which can be summarized as:

- The configuration problem consists of (a) a fixed, pre-defined set of components, where a component is described by a set of properties, ports for connecting it to other components and structural constraints. (b) Some description of the desired configuration by means of user requirements and preferences.
- The configurator must (a) build one or more configurations that satisfy the request if any exist or (b) detect inconsistencies in the requirements.

Furthermore, the authors restrict the configured systems according to some knowledge of the permissible architectures consisting of the set of required functional properties. We can emphasize an important aspect of this definition: it assumes that the set of available components is fixed and completely defined. However in a more general context, abstract descriptions of components may allow a configurator to create them when necessary during the search.

Configuration models

Later researches generalized the concept of permissible architectures to introduce the notion of *configuration models*, where components functional and technical characteristics are defined by their *type*, *attributes* and *relations* to other components. Types and inheritance are used to regroup common characteristics from different components thus obtaining a *taxonomy* of types. Attributes allow the description of characteristics which have domains of primitive types (integers, floats, strings, etc.). Finally, relations to other components are given through *cardinality ports*. We can distinguish composition relations from arbitrary connections. Composition relations are mutually disjoint and express the fact that a component owns sub-components. The set of all relations in a described system is often called its *partonomy*.
The partonomy and taxonomy of a configuration model can be represented by a description logic or in the form of a UML class-diagram [31]. Figure 2.1 shows an example of a PC description in UML.

Constraints

Relations in a partonomy express requirements such as cardinality constraints. However a complex system description usually requires additional knowledge, for instance compatibility of components or restrictions on their attributes depending on the connected objects. For instance, in the previous PC description, we may add a requirement on the sum of consumed energy by each *EnergyConsumer*, such that it is inferior to the energy provided by the *Power* component.
We will survey different formalisms to describe these requirements in Section 2.1.3, but for an easier understanding of configuration problems we base our presentation on the Constraint Programming (CP) approach. CP has quickly become a well-fitted choice for configurators as it offers a completely declarative modelling freedom together with high expressive power.

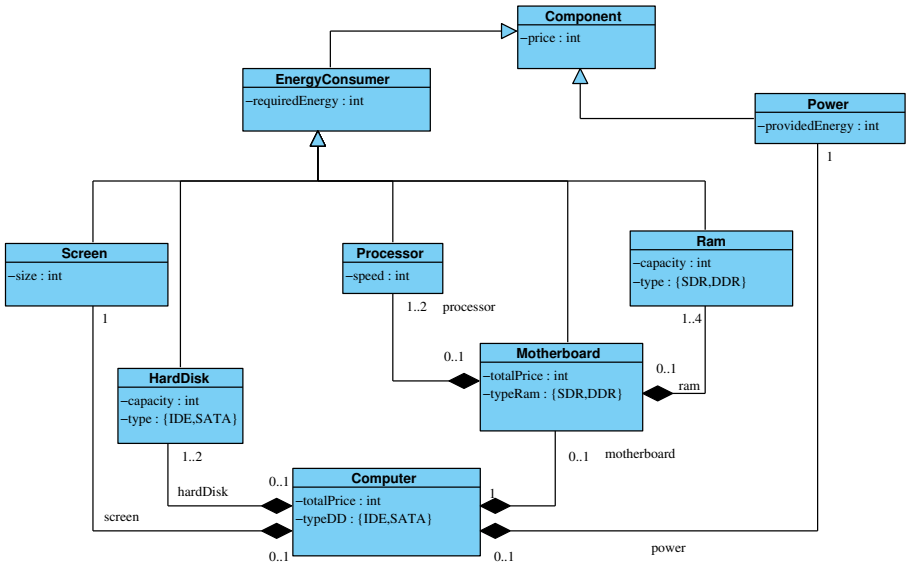


Figure 2.1: A UML object model for a personal computer

A constraint is a boolean assertion on values of variables. Constraints are usually specified using *predicates* (such as equality, inequality, greater-than) on variables. In CP-based configurators, variables may represent the choice of a component's type, the assignment of a value to an attribute, as well as the cardinality and targets of a relation. Therefore constraints can apply to specific components, types, attributes or ports. They can also be specified using compatibility or incompatibility tables on predefined components.
Components involved in a constraint can be predefined components from a catalog, components of a given type, or more generally all components satisfying another given constraint. Since such a set of components may not be completely known during constraint modelling, it calls for the need of *universally quantified* constraints. Here are some informal examples of configuration constraints[1]:

type(component1) = type(component2)
(component1 **isATargetOf** component2.relation1) **implies** type(component1) $\neq$ aType
for all components c **suchThat** c **isATargetOf** component2.relation1: c.attribute1 > 1
for all components c **suchThat** type(c)=aType: c.relation1.cardinality > 1

We present in Section 2.2 a formal language able to express the taxonomy, partonomy and constraints defining a configuration model. Informally, we can describe it as:

[1] we use a dotted notation to access components attributes and relations values

- a set of types T, a set of attributes A, a set of ports P and a set of constraints C that are all mutually disjoint.
- each type $t \in T$ has a set $\mathit{subtypes}(t) \in T$,
- each type $t \in T$ has a set $\mathit{attributes}(t) \in A$ and a set $\mathit{ports}(t) \in P$,
- each type $t \in T$ has a finite domain $D(a, t)$ for each attribute $a \in \mathit{attributes}(t)$,
- each port p has a destination type $\mathit{type}(t, p) \in T$, a minimum cardinality $\mathit{card}_{min}(t, p)$ and an optional maximum cardinality $\mathit{card}_{max}(t, p)$,
- each constraint $c \in C$ is a predicate on elements of T,A and P,
- a (possibly empty) catalog CAT of predefined objects having a type $t \in T$.

A set of components with their types, attributes and ports defined such that all constraints in C are fulfilled is called a configuration or a *model* of the configuration model (unfortunately, the terms "model" in logic and in modelling collide). In a general context, components of the model can either be created or chosen from CAT.

Configuration problem

Based on those configuration models, we can define the associated *configuration problem*:

Definition 2.1.1 *Configuration problem A configuration problem consists in:*

- *a configuration model CM ,*
- *a request R containing a set of requirements (as additional constraints) and/or preferences.*

A configurator or solving procedure *should produce (1) one or more possible models for $CM \wedge R$ if any exist or (2) detect inconsistencies and optionally provide explanations.*

2.1.2 Covered problems and challenges

The expressive power of general configuration models allows to represent a large range of problems. We can split them into two categories: static (or flat) problems describe configurations which have a fixed set of components. Dynamic problems involve optional components or an unbounded number of components.
Static configuration problem are customizations of flat structures of components, or problems with predefined sets of components. These problems can be modelled in such a way that the number of variables participating in the solution is known from start, which allow for solving with various existing techniques for static problems: SAT, Integer Programming, CSP, etc.

In dynamic problems however, the number of variables varies. It can be due to taxonomic reasoning (specializing to a subtype which adds further attributes), to bounded partonomic reasoning (the choice of connected components which may themselves have different subcomponents) or to unbounded partonomic reasoning (connecting an arbitrary number of components). These problems require solvers able to dynamically add variables and constraints during the search.
Configuration has been used by industrials for its ability to solve existing concrete problems in manufacturing and engineering (telecommunication equipment [68], elevators [46], cars, passenger cabins in aircrafts [47], etc.), and later in sales [48] (business-to-customer applications, business-to-business applications). Sales configurators are usually highly interactive and user-driven, whereas a large automatic completion is required for manufacturing. In most of these problems, the set of available components is fixed, made up of concrete elements and taken from a product catalog which is mapped to a constraint model in order to facilitate the maintenance of the system. A limited number of attempts have been made to use configuration in more academic problems where dynamic generation and symbolic reasoning is required: for instance parsing natural language and extracting texts semantics [29, 28].

Challenges

Configurators have to deal with several modelling and problem solving challenges:
Taxonomic and partonomic reasoning induces the dynamic creation of variables and constraints during search.
Unknown and unbounded number of variables require universally quantified constraints. When deciding on relations targets, a configurator has to deal with constraints on set-variables having an open domain.
Search strategies and heuristics need to handle several kinds of decisions: specializing the type variable, selecting the number of connected components, ordering the configuration of components, instantiating attributes, etc.
User preferences also need to be taken into account. Indeed, the potentially huge set of solutions make these crucial to find the preferred configuration.
Component generation is a key issue when the problem does not include a predefined number of components. Generation steps must be added to the search with a careful control in order to avoid cyclic reproduction of an equivalent configuration. The procedure should also avoid the inclusion of unuseful parts.
Explanations of failure may have to deal with large conflicts sets, thus creating the need for a user understandable analysis.
Finally, the combinatorial nature of all existing model search procedures makes of the *scalability* problem a key issue in configuration.

2.1.3 Approaches, Formalisms and Solving methods

The structure of configuration solutions greatly vary depending on the application domain. Configuring a fixed structure induces a fixed number of variables, whereas configuring server racks may imply unbounded number of parts. Those differences explain why so many different techniques have been applied to configuration. Surveys can be found in [91, 47, 103, 57]. They classify the different approaches into rule-based reasoning [67], case-based reasoning [80] and model-based reasoning [75]. The latter one covering description logics [76, 68], constraint programming [74, 33] and resource models [56]. We can add knowledge based approaches [103], logic programming [97], object-oriented approaches [65, 103], SAT [94], binary decision diagrams [55], and answer-set programming [93].
The contributions of these researches reside either in the representation of configuration models or in the model search procedure (i.e the solving method). Most of the presented methods require a translation from configuration models defined in a higher level language to the chosen solving formalism.

Rules

Rule-based approaches express requirements in terms of assertions mapped to components through rules. When a rule is fired, it can generate new components or assign a value to existing components attributes. This modelisation tends to produce multiple rules with complex conditions for the same functionality. Such a system is not modular, since a change in the model may imply modifications in several rules. This caused a severe maintenance problem in the R1/XCON configurator [67, 7].
Furthermore, a rule based configurator is an *incomplete* solving method since not all combinations are explored. Therefore its unability to discover a solution does not imply that a solution does not exist. As a consequence they cannot be used for proving the non-satisfiability of a configuration problem (even in the case where only discoverable finite solutions are taken into account for satisfiability).

Case-based

Case-based methods save knowledge about already solved problems in order to find solutions to similar problems. For instance a reference configuration can be adapted with minor modifications. Although case-based methods have proven useful in repetitive situations [80], they usually provide bad-quality solutions as their conservative aspect does not favor original configurations.

Model-based

Model-based reasoning separates the problem description from the solving method. As it facilitates maintenance issues, it quickly became the default choice for configurators, giving birth to many paradigms. We present in the following the main approaches.

Most of them are partial techniques answering specific configuration issues and have to be combined to offer a full configuration solution.

Resource-based

Resource-based approaches are useful for systems where components provide and consume resources. For instance, in the PC configuration, a component provides power which is consumed by other devices. Configurators such as [56] use resource balancing in the selection of components. Although resources are frequent in configured systems, the approach needs to be combined with other techniques.

Knowledge compilation

Preprocessing techniques can be very efficient to lower the computational complexity. Depending on the application domain, the configuration knowledge can be represented using binary decision diagrams [55], automatons [5], or the decomposable negation normal form [21]. Other techniques include synthesis trees [112] and cluster trees [23, 82]. Knowledge compilation has been successfully applied to car configuration [55, 82]. However it relies on properties of the modelled domain and thus cannot be applied to all configuration problems.

Description Logics

Description logic is a family of logic based knowledge representation formalisms, descendants of KL-ONE, frame-based systems and semantic networks. It is made of decidable fragments of first-order logic. A domain is described in terms of concepts (classes), roles (properties, relationships) and individuals. Description Logic allows to describe and reason on taxonomies and partonomies having complex semantics such as relations specializations or concepts intersections. DL expressiveness is not usually sufficient for configuration knowledge as it does not handle numerical requirements for example. It is combined with rules or constraints in hybrid approaches like the PLAKON-project [20] or the CLASSICS-project [68].

Logic programming

In [97], configuration knowledge is represented using a *weight constraint rule* language. The approach extends logic programming with cardinality and resource constraints support. The rules semantics are captured by *stable models*. Their main property is to be models that are *justified* by the rules, therefore avoiding unuseful components in the proposed solutions. The resulting configuration task is proven to be NP-Complete. This computational property is due to the fact that only fixed, predefined sets of available components are considered. The limitation thus discards its use for problems involving, for instance, generic descriptions of an arbitrary number of components. It is also acknowledged by the authors that the proposed formalism is not well-fitted as a high-level

description language. A translation is thus first required, which is however shown to be modular.

Constraint Programming

We have already shown how constraints are well-suited to define requirements in a configuration problem. The next paragraphs will cover the different formalisms that have been adapted or developed for configuration upon the use of constraints.

CSP The Constraint Satisfaction Problem (CSP) consists in assigning values to variables which are subject to a set of constraints.

Definition 2.1.2 (Constraint Satisfaction Problem) *A CSP is defined by a triple (X,D,C) where:*

- *X is a finite set of variables* $\{X_1, \ldots, X_n\}$
- *D is a finite set of domains* $\{D_1, \ldots, D_n\}$ *where* D_i *is a set of possible values for* X_i
- *C is a finite set of constraints where each constraint is an assertion on a subset of X* $\{X_j, \ldots, X_k\}$ *defined by a subset of* $D_j, \ldots, D_k$

A solution of the CSP is a total assignment satisfying each of the constraints.
CSP are well-suited to represent static configuration problems where all the variables are known from the start. For instance, the set of components $O = \{O_1, \ldots, O_l\}$ participating in the solution is fixed, each port has a bounded cardinality, and the classification of a component does not imply additional attributes or relations. In such a CSP, the set of variables X can be obtained with:

- a variable $X_{t_{ij}}$ for the type of each component O_i, with a finite domain $D_{t_{ij}}$ of all possible types
- a variable $X_{a_{ij}}$ for each attribute $a_j \in \mathit{attributes}(O_i)$, with a finite domain $D_{a_{ij}}$ of possible attribute values
- a variable $X_{p_{ijk}}$ for each possible target O_k for each port $p_j \in \mathit{ports}(O_i)$, with a boolean domain $D_{p_{ijk}}$

As the set of components is fixed, constraints can be given in extension and there is no need for universally quantified constraints. However, issues are raised when one tries to express complex constraints, i.e constraints on types or constraints which traverse components relations.
As an alternative to CSPs, one can use SAT solvers based on propositional logic for the same range of problems.

Conditional CSPs Conditional CSPs were introduced in [75]. Called dynamic CSPs at the beginning, they were later renamed to conditional CSPs. They deal with optional variables and constraints which are activated depending on given conditions. Optional variables have an attached boolean *activity* variable. When the activity variable is true, the attached variables are part of the solution. A constraint is active only if all its variables are active.
Conditional CSPs are well-suited for some configuration problems which cannot be represented with a classical CSP. If a taxonomic reasoning induces a new attribute a_j when a component is specialized to the type t_i, it can be modelled as an optional variable: a constraint implies that the activity variable attached to a_j is true when the type variable takes value t_i.
Conditional CSPs can also deal with systems where the choice of a complex component versus a simple one induces the need to select a set of sub-components. In the same way, an activity variable is attached to the complex component variables representing the ports to the sub-components.
Those problems can also be formalized using another variant of CSPs: Composite CSPs [90] where the variable domains may contain sub-CSPs.
Conditional and composite CSPs have several limitations. They suppose either an automatic translation of configuration models or a (sometimes difficult) modelling effort. They also induce a large increase in the number of variables and constraints. Furthermore, the whole set of possible optional components needs to be known from start, limited in size, thus preventing for instance the creation of components upon necessity.

Generative CSPs Generative CSPs [33, 104] address the restriction of having an explicit set of available components before the beginning of the search. An *instance-set* is used for each possible type. It is represented by a set variable with an open domain. When the lower bound of a port's cardinality is increased, a generative constraint adds an instance to the instance-set of the port's target type. The ability to dynamically generate components allows generative CSPs to handle most configuration problems. However they still face the difficulty of expressing complex constraints over types. [36] is an approach where configuration knowledge, as predicate logic extended with set constructs, is mapped to a generative CSP.

Object-oriented

Object-oriented configurators [65] mix techniques from description logics and constraint programming. Configurations are defined as instances of an object model. On the one hand, description logics allow to describe object models through class-based taxonomies and partonomies. They also provide means for classification and objects creation. On the other hand, constraint programming offers high expressiveness and fitted problem-solving methods like generative CSPs. The combined language takes its predicates from the constraint language and its terms are made of symbols from the description logics vocabulary as well as logical variables.
Object-oriented configurators combine modelling freedom from object models and con-

straints together with solving efficiency. Furthermore, solving is done directly at the level of the modelling language.
We choose this approach for defining our configuration problems throughout this thesis. We present a formalization and generalization of this representation as *constrained object models* in the next Section, together with associated finite model semantics and solving algorithms.

2.1.4 Position in AI

The use of configuration as a tool for symbolic computation is a relatively recent field of research. A number of different formalisms and corresponding search engines were developed to address configuration problems in the industry or research, but did not lead to a commonly accepted configuration language, as was the case for CSPs in constraint programming. Also configuration is mainly used in industrial applications, and therefore lacks a well-established position in theoretical AI. We now try to give our intuition of why and how configuration can be used as a tool for symbolic reasoning in AI problems.
Configuring requires a formal description of the set of all viable composite constructs that can be realized according to constraints. Such a description is called a "model" of a "world", or "universe". Configuring then amounts to finding an instance of the configuration model plus a query. If successful, the existence of this instance (a finite model, unfortunately the term "model" is used in both contexts) proves that the logical conjunction of a theory (the "model" of the world) and a query is satisfiable. This clearly means that configuration can be viewed as a theorem proving method. Indeed, finite model search is already used in the theorem proving community to exhibit counter-examples ($T \models F$ is proven false by finding a model for $T \wedge \neg F$), either to disqualify conjectures (for instance inequational theories [8]), or locally to simplify syntactical proofs in automated theorem provers.
From the expressivity viewpoint, general configuration models, which make use of a subset of description logic, obviously involve first (or higher) order logic statements. More specifically, real world models require dealing with predicates, universal quantifications ("all cars have wheels"), a complex form of existential quantifications through relations cardinalities ("between four and six wheels"), and concept hierarchies. Formal languages able to deal with such constructs start at the level of first order logic, but in fact require more, since it must often be dealt with sets, or quantifying over relations. Concept hierarchies and finite domain attributes also occur in most models, which introduces symbolic reasoning.
Now at the complexity level, it can be seen that configuring, viewed as the search for a finite model in the adequate logic, is a task that may never terminate. This is so when the problem has no solution, because the search space is potentially infinite: objects can connect to different objects indefinitely depending upon the model. But this is also the case for some satisfiable problems that happen to only have infinite models. An idea of this possibility can be sketched by saying that the model of reality where each man has a father, and there is a man, is a configuration problem. Figure 2.2 illustrates this. In many cases however, a configuration problem can be modelled in order to yield a *finite*

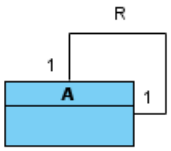


Figure 2.2: A (UML) object model which only accepts infinite models (the transitive closure of the relation R is irreflexive)

search space. Under such settings, the problem becomes *NP-complete*.
The structures of the configurations created (interconnected components) can be viewed as colored graphs. Thus configuration shares many concepts and properties from the *graph theory* and *graph generation* procedures, and is amenable to similar symmetry breaking approaches. How these can impact search methods will be further discussed in Chapter 5.
Configuration also shares similarities with *planification*. Planification concerns the definition of actions sequences able to achieve a goal or final state. Both paradigms have to deal with the dynamic generation of interconnected objects.
If we consider solving methods, configuration falls within the scope of semantic approaches. Under the condition of finite search space, correct and complete configuration algorithms can be implemented with an enumerative search of possible combinations, thus placing it in the field of combinatorial search procedures. In particular, the use of constraint programming techniques makes it an extension of the Constraint Satisfaction Problem (CSP) to first-order logic. The solving methods share many similarities with CSPs: the problem can be seen under the viewpoints of satisfaction or optimization, and exhaustive or incomplete algorithms can be used. An original incomplete stochastic framework is proposed in Chapter 6.

2.2 A Z-based formalisation for constrained object models

There is no commonly agreed formalisation for object models. The modeling language is usually dependent on the application domain, or on the solving tool. In object-oriented modelisation, taxonomies and partonomies are often represented using UML class diagrams[2], coupled with OCL[3] for additional constraints. However UML+OCL, to a large extent, lack formal foundations as well as expressive power. We choose to formally specify our constrained object models using the *Z* relational language. Relational languages are second order formalisms involving set theoretic constructs. The choice of Z takes place in the work initiated by Laurent Henocque in [49] about using *Z* schemas, and

[2]A large documentation on UML is available on the OMG website at http://www.omg.org/technology/documents/formal/uml.htm
[3]Object Constraints Language - last version : http://www.omg.org/docs/ptc/03-10-14.pdf

followed by Mathieu Estratat in [28]. Here are the main motivations:

- Z offers a common language for classes, relations and constraints.
- it is at least as expressive as predicate logics.
- its formal foundations allow for proofs and reproducible results.
- it is independent from the solving tool.
- we follow the effort on pushing Z as a shared language for the community.

In the following, we give a full presentation of constrained object models specified in Z. Each objet model element is illustrated using the UML notation [45]. We also provide finite models semantics for such COMs, and discuss the implementation in configuration tools.

2.2.1 Introducing Z

It is difficult to make this Section self contained, since this would suppose a thorough presentation of both the UML notation [45], and the Z specification language [101]. The reader, if novice in these domains, is kindly expected to make his way through the documentation, which is electronically available. However, we provide a brief introduction to useful Z constructs and notations.

data types as named sets

Z data types are possibly infinite sets, either uninterpreted ($DATE$), or axiomatically defined as finite sets (dom), or declared as free types ($colors$) :

$$[DATE]$$

$$dom : \mathbb{F}\,\mathbb{N}$$

$$colors ::= red \mid green \mid blue$$

All other relation types can be built from cross products of other sets.

axiomatic definitions

Axiomatic definitions allow to define global symbols having plain or relation types. For instance, a finite group is declared as:

$$\begin{array}{l} zero : dom \\ inverse : dom \longrightarrow dom \\ sum : (dom \times dom) \longrightarrow dom \\ \hline \forall\, x : dom \bullet sum(x, inverse(x)) = zero \\ \forall\, x : dom \bullet sum(x, zero) = x \\ \forall\, x, y : dom \bullet sum(x, y) = sum(y, x) \\ \forall\, x, y, z : dom \bullet sum(x, sum(y, z)) = sum(sum(x, y), z) \end{array}$$

The previous axiomatic definition illustrates cross products and function definitions as means of typing Z elements. Now axioms or theorems are expressed in classical math style, involving previously defined sets. For instance, we may formulate that the inverse function above is bijective (this is a theorem) in several equivalent ways as e.g.:

$$inverse \in dom \rightarrowtail\!\!\!\!\!\rightarrow dom$$

($\rightarrowtail\!\!\!\!\!\rightarrow$ defines a bijection), or explicitly using an appropriate axiom :

$$\forall\, y : dom \bullet \exists_1\, x : dom \bullet inverse(x) = y$$

schemas

The most important Z construct, *schemas*, occur in the specification in the form of named axiomatic definitions. A schema $[D \mid P]$ combines one or several variable declarations (in the declaration part D) together with a predicate P stating validity conditions (or constraints) that apply to the declared variables. The reader is directed to the Z Reference Manual [101] for details.

SchemaOne

$a : \mathbb{N}$
$b : 1 \mathinner{..} 10$

$b < a$

The schema name hides the inner declarations, which are not global. A schema name (*SchemaOne*) is a shortcut for its variable and predicate declarations that can be universally or existentially quantified at will. Schemas do not define object identity hence are not suitable as such to describe object oriented semantics.

2.2.2 Constrained Object Models

Constrained object models allow for the specification of problems where occur inheritance relationships, relations in the usual sense, sets and bags of various component types, and constraints applied to all these elements.

Definition 2.2.1 (Constrained Object Model) *A Constrained Object Model (COM) is a set of classes and relations subject to well-formedness constraints.*

Objects* and *Classes

The most general type is *Object*. We define "objects" using a non interpreted Z type. No specific semantic attaches to this definition of objects, beyond "identity", later enforced by adequate finite model semantics.

$$[Object]$$

Class is the set of finite subsets of [*Object*]. This is specified using an alias notation.

$$Class == \mathbb{F}\ Objet$$

A class "A" simply defines as a symbol having the type *Class*. It hence denotes a finite subset of the set of objects. The choice that classes are finite is a pragmatic one within the scope of use of this specification.

$$A : Class$$

Figure2.3 represents the UML visual notation of such a class.

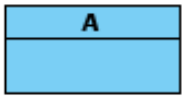


Figure 2.3: UML notation for class A

Class attributes

A class "attribute" is defined as a function mapping class instances to a domain of values. Finite numeric domains are available as well as symbolic domains, like character strings (*String*). *String* is defined as an uninterpreted data type, like *Object* above.

$$[String]$$

For instance, let *att* be an attribute for class A, with $\mathbb{N}$ as a domain:

$$att : A \longrightarrow \mathbb{N}$$

In UML, class attributes are presented within the class body, as in Figure 2.4.

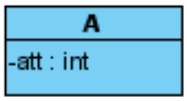


Figure 2.4: UML notation for an attribute

Associations

The Z notation for an association relationship *ArelB* between two classes A and B is as follows:

$$ArelB : A \leftrightarrow B$$

Since we treat classes as the sets of their instances, the definition of relations in Z strictly matches the specification of relations in object models. *ArelB* is a set of tuples: $A \leftrightarrow B = \mathbb{P}(A \times B)$.

Roles

Binary relations can be defined (or the specification can be complemented) using one or both of its opposite roles. A role is a function mapping each instance of its source type to the set of its "connected" instances of the distant type. Roles make explicit the function allowing to navigate through a relation link. Roles are used as the basic implementation of binary relations in some systems.

Definition 2.2.2 (Role) *Let A and B be two classes, and $ArelB$ a relation among them. The corresponding roles (here called $roleAB$ and $roleBA$) are defined as:*

$$roleAB : A \longrightarrow \mathbb{P}\, B$$
$$\forall\, a : A \bullet roleAB(a) = ArelB(\!|\{a\}|\!)$$

$$roleBA : B \longrightarrow \mathbb{P}\, A$$
$$\forall\, b : B \bullet roleBA(b) = ArelB^{\sim}(\!|\{b\}|\!)$$

The operator $(\!|\_|\!)$ is the relational image ([101] p.101). The operator $^{\sim}$ denotes the relational inverse.

Role multiplicity

Modeling languages have made popular the possibility for a model to constrain the number of participants in a relation. There are a limited number of syntaxes and semantics for these constraints. We strictly match UML definitions. The multiplicity specification at the end of a role counts how many target instances can exist for each instance in the role source. The cardinality operator in Z ($\#$) can be used straightforwardly to implement such constraints. For instance, if we consider the elements in Figure 2.5:

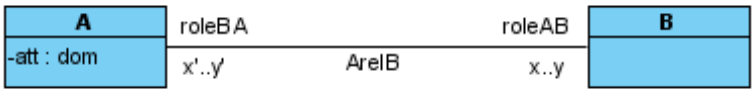


Figure 2.5: UML notation for an association

$$\forall\, a : A \bullet x \leq \#(roleAB(a)) \leq y$$
$$\forall\, b : B \bullet x' \leq \#(roleBA(b)) \leq y'$$

Composition relationships

Unlike a generic association, a composition association involves implicit constraints. There are variants to the semantics of compositions. We address here the most largely accepted semantics, according to which "components" cannot belong to several distinct "composites". Composition associations are drawn in UML as in Figure 2.6 using a black diamond on the "composite" role. With Z, it can be specified by saying that the relation role mapping "components" to their "composite" is a function (when components cannot exist on their own) or a partial function (if they can):

$$C : Class$$
$$D : Class$$

$$CcompoD : C \longrightarrow \mathbb{P}\, D$$
$$\forall\, c : C \bullet x \leq \#(CcompoD(c)) \leq y$$

We then need to ensure that the target of a composition relation cannot be shared by multiple sources. Although it can be achieved in various ways, we present here a solution based on an explicit inverse role, frequently modelled in configuration problems:

$$CcompoDrev : D \rightarrowtail C$$
$$\forall\, c : C;\ d : D \bullet d \in CcompoD(c) \Leftrightarrow c = CcompoDrev(d)$$

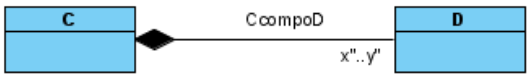


Figure 2.6: UML notation for a composition relation

Inheritance

Inheritance relationships allow for defining features common to a group of classes without needing to redefine these features multiple times. Standard software engineering practices consider that the inheritance relationship among classes should strictly match a set inclusion relationship on the corresponding sets of instances. We obey these requirements. Now considering that the class A from Section 2.2.2 inherits a class C, we have :

$$A \subseteq C$$

One usually expect disjoint (non overlapping) subclasses semantics (in UML the choice of having overlapping subclasses is left to the designer). If now the class B inherits C, we have :

$$B \subseteq C$$

And if we choose to enforce non overlapping (the most common case) :

$$\mathsf{disjoint}\ \langle A, B \rangle$$

The corresponding UML notation for inheritance is presented in Figure 2.7 . If an

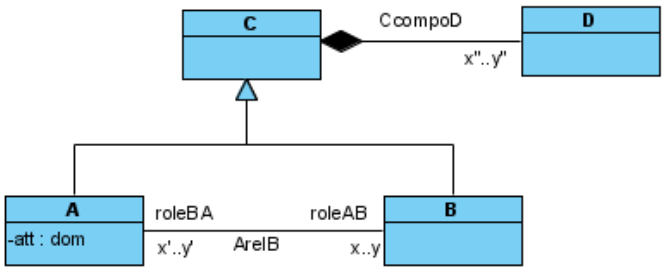


Figure 2.7: UML notation for inheritance

intermediate class in an inheritance hierarchy is *abstract* (i.e. it cannot have instances of its own, that would not be a member of any of its subclasses), we may use the Z *partition* operator:

$$\langle A, B \rangle\ \mathsf{partition}\ C$$

This forms the main body of a straightforward method for formally specifying constrained object models using the Z language. Of course "constrained object models" involve "constraints". The next section deals with them.

Constraints

Again, we propose the use of Z here, with the following arguments:

- Z is a second order, set theoretic language with an extremely high expressive power,
- being extensible, Z allows for the statement of inline operators that help achieving readability

A constraint is formulated in Z as a logical predicate (page 67 in [101]) applied to elements of type $\mathbb{N}$, $\mathbb{Z}$, *String*,*Object*, $\mathbb{F}$ *Object* and $[\![Object]\!]$. Usual operations (plus a rich library) over naturals or sets are available. A full list is given in the *Mathematical Tool-kit* starting p.86 in [101]. Infix de-referencing operators in the spirit of [49] can be specified to traverse relationships in the same way as OCL does. Such operators help making constraints more readable by left to right traversal (instead of recursive embedding).

De-referencing operators

- .: applies to a single instance, to yield a set or a singleton. It is used to reach (the value of) an *Object*'s attribute
- $\rightarrow$: applies to a set, to yield a set or a singleton. This is mainly the same as . applied to a set.
- $\rightsquigarrow$: applies to a set, to yield a *bag*[4]. It is used to traverse an attribute or a relation, and remain aware of repetitions.

These three (groups of) operators are formally generically specified in Table 2.1. Their use allows for more readable constraints.

Defining generalized operators Z is extensible, and allows for the definition of language extensions. In many constrained object models, the situation arises where one must compute the sum of the values of an attribute collected as a bag (for instance the prices of components in a construction). Such bags of prices may indeed involve repetitions, that must not be lost. As an example we define the *bagsum* operator.

$$
\begin{array}{|l}
bagsum : (\mathbb{N} \longrightarrow \mathbb{N}) \longrightarrow \mathbb{N} \\
\hline
bagsum(\varnothing) = 0 \\
\forall\, b : \mathbb{N} \longrightarrow \mathbb{N} \bullet \forall\, x : \mathrm{dom}\, b \bullet bagsum(b) = x * b(x) + bagsum(b \uplus \{x \mapsto b(x)\})
\end{array}
$$

The symbol $\uplus$ is the bag difference operator (p.126 in [101]).
For example, consider the object model of Figure 2.8 and the constraint specifying that for each motherboard, the sum of RAM capacity must be superior or equal to 512. We can write it:

$$\forall\, cm : CarteMere \bullet bagsum(cm.ram \rightsquigarrow capacity) \geq 512$$

alternately, we can use the operator $\rightarrow$ defined in Table 2.1:

$$\forall\, cm : CarteMere \bullet (cm.ram \rightsquigarrow capacity) \rightarrow bagsum \geq 512$$

[4]A bag is a counted set where the same element can be repeated.

$$
\begin{array}{l}
[U, X] \\
\hline
\_.\_ : (U \times (U \leftrightarrow X)) \longrightarrow \mathbb{P}\, X \\
\_.\_ : (U \times (U \longrightarrow X)) \longrightarrow X \\
\_ \rightarrow \_ : (\mathbb{P}\, U \times (U \leftrightarrow X)) \longrightarrow \mathbb{P}\, X \\
\_ \rightarrow \_ : (\mathbb{P}\, U \times (U \longrightarrow X)) \longrightarrow \mathbb{P}\, X \\
\_ \rightarrow \_ : (\mathbb{P}\, U \times (\mathbb{P}\, U \longrightarrow X)) \longrightarrow X \\
\_ \rightsquigarrow \_ : (\mathbb{P}\, U \times (U \leftrightarrow X)) \longrightarrow \mathrm{bag}\, X \\
\_ \rightsquigarrow \_ : (\mathbb{P}\, U \times (U \longrightarrow X)) \longrightarrow \mathrm{bag}\, X \\
\hline
\forall\, e1 : U \;;\; e2 : (U \longrightarrow X) \bullet e1.e2 = e2(e1) \\
\forall\, e1 : U \;;\; e2 : (U \leftrightarrow X) \bullet e1.e2 = e2(\!|\{e1\}|\!) \\
\forall\, e1 : \mathbb{P}\, U \;;\; e2 : (U \longrightarrow X) \bullet e1 \rightarrow e2 = \\
\qquad \{x : U \mid x \in e1 \bullet e2(x)\} \\
\forall\, e1 : \mathbb{P}\, U \;;\; e2 : (U \leftrightarrow X) \bullet e1 \rightarrow e2 = e2(\!|e1|\!) \\
\forall\, e1 : \mathbb{P}\, U \;;\; e2 : (\mathbb{P}\, U \longrightarrow X) \bullet e1 \rightarrow e2 = e2(e1) \\
\forall\, f : U \longrightarrow X \bullet \varnothing \rightsquigarrow f = [\![]\!] \\
\forall\, f : U \longrightarrow X \;;\; s : \mathbb{P}_1\, U \bullet \forall\, u : s \bullet s \rightsquigarrow f = (s \setminus \{u\}) \rightsquigarrow f \uplus [\![f(u)]\!] \\
\forall\, f : U \leftrightarrow X \bullet \varnothing \rightsquigarrow f = [\![]\!] \\
\forall\, f : U \leftrightarrow X \;;\; s : \mathbb{P}_1\, U \bullet \forall\, u : s \bullet s \rightsquigarrow f = (s \setminus \{u\}) \rightsquigarrow \\
\qquad\qquad f \uplus \{x : X \mid (u, x) \in f \bullet x \mapsto 1\}
\end{array}
$$

Table 2.1: Z: defining de-referencing operators

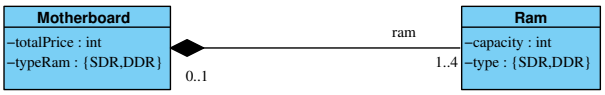


Figure 2.8: UML submodel for a personal computer

2.3 Finite Model Semantics for Constrained Object Models

We define the semantics of constrained object models by specifying an interpretation function. This requires to give an interpretation of any meaningful elements such as objects, classes, attributes and relations.

Definition 2.3.1 (Interpretation of a constrained object model) *The interpretation $I(M)$ of a constrained object model M is defined by the recursive generalization of its elements interpretation. The elements classes, objects, attributes, roles and relations are given their own interpretation; whereas constraints have the usual interpretation of predicate logic on those elements.*

2.3.1 Interpretation of COMs elements

The interpretation of a constraint $I(ct)$ is defined by the Z notation: it is the interpretation of predicate logic. [28] details the interpretation of the other introduced elements: *Objects*, *Strings*, *Classes*, *attributes* and *relations*.

Interpretation of *Object*, *String* and *Class* I defines a projection of non-interpreted types on a finite set of integers. All objects are mutually distinct. The interpretation function associates a different integer to any element of type *Object*. This function is injective (represented by the symbol $\rightarrowtail$).

$$IObj : Object \rightarrowtail \mathbb{N}$$

Any element of type *String* is also interpreted as an integer.

$$IString : String \rightarrowtail \mathbb{N}$$

The interpretation of an integer is the integer itself: $\forall\, n : \mathbb{N} \bullet I(n) = n$. The interpretation of the other elements naturally stems from I: By definition, any *Class* is a finite set of *Objects* and I thus associates a finite set of integers.

$$IClass : Class \rightarrowtail \mathbb{F}\, \mathbb{N}$$
$$\forall\, c : Class \bullet IClass(c) = IObj(\!|c|\!)$$

Interpretation of attributes Any attribute is interpreted as a couple (integer interpreting the object, interpretation of the attribute value) Attribute of type integer:

$$IAttN : (Object \longrightarrow \mathbb{N}) \rightarrowtail \mathbb{F}(\mathbb{N} \times \mathbb{N})$$

$$\forall\, att : (Object \longrightarrow \mathbb{N}) \bullet IAttN(att) = \bigcup\{o : Object \mid o \in \mathrm{dom}\, att \bullet \{(IObj(o), att(o))\}\}$$

Attribute of type String:

$$IAttString : (Object \longrightarrow String) \rightarrowtail \mathbb{F}(\mathbb{N} \times \mathbb{N})$$

$$\forall\, att : (Object \longrightarrow String) \bullet IAttString(att) = \bigcup\{o : Object \mid o \in \mathrm{dom}\, att \bullet \{(IObj(o), IString(att(o)))\}\}$$

Interpretation of arbitrary relations Any arbitrary relation is interpreted as a couple of integers:

$$IRel : (Object \leftrightarrow Object) \rightarrowtail \mathbb{F}(\mathbb{N} \times \mathbb{N})$$

$$\forall\, rel : (Object \leftrightarrow Object) \bullet IRel(rel) = \bigcup\{o_1, o_2 : Object \mid (o_1, o_2) \in rel \bullet \{(IObj(o_1), IObj(o_2))\}\}$$

Interpretation of roles and composition relations Any role is interpreted as a set of couples (integers, set of integers):

$$IRole : (Object \longrightarrow \mathbb{P}\, Objet) \rightarrowtail \mathbb{F}(\mathbb{N} \times \mathbb{F}\, \mathbb{N})$$

$$\forall\, role : (Object \longrightarrow \mathbb{P}\, Object) \bullet IRole(role) = \bigcup\{o : Object \mid o \in \mathrm{dom}\, role \bullet \{(IObj(o), IObj(\!|role(o)|\!))\}\}$$

Interpretation of composition relations As a role, a composition relation is a function associating a set of objects to an object. Their interpretations are similar. The reverse partial function of a composition relation is equal to a set of couples (integer,integer):

$$ICompoRev : (Object \nrightarrow Objet) \rightarrowtail \mathbb{F}(\mathbb{N} \times \mathbb{N})$$

$$\forall\, comporev : Object \nrightarrow Object \bullet ICompoRev(comporev) = \bigcup\{o : Object \mid o \in \mathrm{dom}\, comporev \bullet \{(IObj(o), IObj(comporev(o)))\}\}$$

Based on the above interpretations and predicate logic, we obtain a total interpretation of a constrained object model. An interpretation satisfying all the constraints of a constrained object model is called an *instance* or *a model*.

Definition 2.3.2 (Finite model) *A finite model is a finite set of objects satisfying the set of constraints defined on them.*

Definition 2.3.3 (Constraint satisfaction) *a constraint* ct *if satisfied if its interpretation I is true (I(*ct*) = true).*

2.3.2 Interpretation example

We describe an example of COM interpretation, first presented in [28]. We consider the object model of Figure 2.9 (it is a submodel of Figure 2.1), and the following constraints:

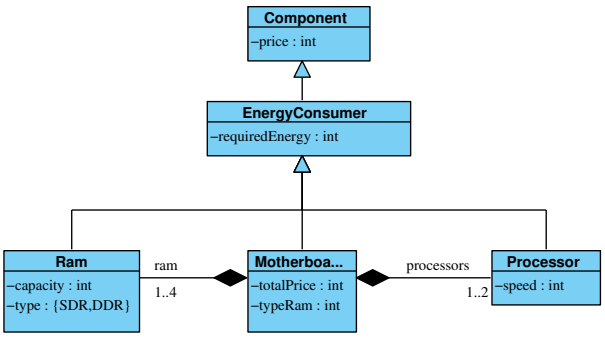


Figure 2.9: A UML object model of a motherboard

$$\forall\, mb : Motherboard\ ;\ r : Ram \mid r \in mb.ram \bullet mb.typeRam = r.type$$
$$\forall\, mb : Motherboard\ ;\ r : Ram\ ;\ p : Processor$$
$$\quad \mid r \in mb.ram \wedge p \in mb.processor$$
$$\quad \bullet\ mb.totalPrice =$$
$$\qquad mb.price + bagsum((mb.ram) \rightsquigarrow price) + bagsum((mb.processor) \rightsquigarrow price)$$
$$\forall\, mb : Motherboard \mid \#(mb.processor) = 2 \bullet$$
$$\quad \forall\, p_1, p_2 : Processor \mid p_1 \in mb.processor \wedge p_2 \in mb.processor \bullet$$
$$\qquad p_1.speed = p_2.speed$$

We consider the following set of finite objects E_f:
$E_f = \{mb_1 : motherboard, p_1 : Processor, r_1 : Ram\}$ such that:

1. $mb_1.ram = \{r_1\}$, $mb_1.processor = \{p_1\}$,

2. $mb_1.totalPrice = 205$, $mb_1.typeRam = SDR$, $mb_1.requiredEnergy = 205$, $mb_1.price = 80$

3. $r_1.capacity = 256$, $r_1.type = SDR$, $r_1.requiredEnergy = 5$, $r_1.price = 50$

4. $p_1.speed = 2000$, $p_1.requiredEnergy = 10$, $p_1.price = 75$

Interpretation of *objects* and *classes* For readability reasons, we suppose that the 3 objects of this example are associated to numbers 1, 2 and 3.

$$IObj(mb_1) = 1$$
$$IObj(r_1) = 2$$
$$IObj(p_1) = 3$$

Classes interpretation:

$$IClass(motherboard) = \{1\}$$
$$IClass(Ram) = \{2\}$$
$$IClass(Processor) = \{3\}$$
$$IClass(Composant) = \{1, 2, 3\}$$
$$IClass(EnergyConsumer) = \{1, 2, 3\}$$

DDR and SDR are two elements of type String, their interpretation is:

$$IString(SDR) = 1$$
$$IString(DDR) = 2$$

Interpretation of composition relations

$$IRole(ram) = \{(1, \{2\})\}$$
$$IRole(processor) = \{(1, \{3\})\}$$

Interpretation of attributes

$$IAttN(totalPrice) = \{(1, 205)\}$$
$$IAttString(typeRam) = \{(1, 1)\}$$
$$IAttString(type) = \{(2, 1)\}$$
$$IAttN(capacity) = \{(2, 256)\}$$
$$IAttN(speed) = \{(3, 2000)\}$$
$$IAttN(requiredEnergy) = \{(1, 205), (2, 5), (3, 10)\}$$
$$IAttN(price) = \{(1, 80), (2, 50), (3, 75)\}$$

This interpretation satisfies all the constraints, therefore it is an instance of the constrained object model. Figure 2.10 describes the instance using UML.

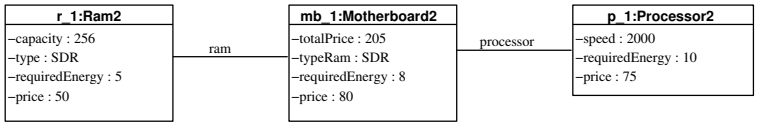


Figure 2.10: A possible solution for the motherboard sub-model

2.4 Finite model search algorithms for constrained object models

In this thesis, we will consider algorithms based on an enumerative search of finite models which have proven efficient for constraint programming problems. We sketch a potential algorithm:

- the configurator starts with a *root* object (in the absence of a root object, it is always possible to add one to the COM without loss of generality),
- the configurator enumerates for this object the possible values for classification, attributes, relations cardinality, and selection (or creation) of the relations targets.
- targets (dynamically) added to the instance are treated in a recursive way until all objects are completely configured,
- the configurator *backtracks* to the next value when it cannot expand the instance without violating the constraints,
- whenever an instance satisfies all of the model and user constraints, it is a solution of the problem.

Definition 2.4.1 (Configuration space) *A configuration space is the set of all potential combinations for a configuration model.*

If the configuration space is finite, an enumerative search can be done in finite time. Although this is not the general case of configuration, there are several methods to guaranty this property, among which:

- upper bounds can be given to either the total number of objects or to the number of objects per class,
- the COM contains no cycles in its partonomy and no unbounded cardinalities,
- the objects participating in the solution cannot be created by necessity but have to be selected from a finite catalog.

2.4.1 Search enhancements

In a similar way as CSPs, enumerative configuration algorithms can benefit from several techniques: heuristics for choosing variables and values, constraints propagation for reducing variables domains, symmetry breaking, etc. However applying them to first-order theories is far from obvious and raises several questions already overviewed in Subsection 2.1.2. We will extensively treat the question of symmetry breaking for configuration in Chapter 5. The search for a model can also be done under the scope of

satisfiability (finding any model) or optimization (finding the best model with respect to an optimization factor).
Incomplete algorithms, a promising field of research in CSPs, can also be applied for finding finite models in configuration. These algorithms only partially explore the configuration space with several methods in order to reach a solution. In particular, stochastic methods use a combination of random moves and heuristics. To the best of our knowledge, such algorithms have not been developed yet for configuration. We will propose an original stochastic algorithm based on ant colonies behaviour in Chapter 6.

2.4.2 JConfigurator

JConfigurator is a java configurator for object oriented configuration. It relies on a combination of description logics and a constraints language, close to FPC [64]. The search is based on the generalized tableau method. [58] is an article devoted to JConfigurator logics and algorithm.
Jconfigurator offers a high expressivity for constraints with competitive computational results. We chose it for the implementation of our Semantic Web Services composer presented in Chapter 4.
In [28], Mathieu Estratat proposes a modular translation of COMs expressed in the Z language to JConfigurator. The translation is claimed correct and complete.
We provide an example of COMs in JConfigurator in Annex I. The presented constraints are the counterpart of the Z-model presented in Chapter 4.

2.5 Conclusion

In this chapter we have presented the logical paradigm of configuration, its main modelling and solving challenges, as well as possible applications. We also described a generalization based on finite models search for constrained object models which offers a high degree of expressivity. The notation for COMs will be used throughout the rest of the document.
Configuration applications are mostly restricted to manufacturing where configured architectures are made of concrete and predefined components. Many AI problems require to reason about concepts or symbolic objects where it is needed to create them dynamically from abstract descriptions. Although seldom considered, we wish to support the claim that configuration is a viable option, in particular for problems of first-order theories which are hardly solved in existing frameworks. Moreover the object-oriented and declarative nature of the presented formalism allows to easily express problem specifications close to the associated application domain,. Whereas most other AI methods require translations to their logical formalisms, object-oriented configuration solving is done directly at the level of the modelling language. An example of such an AI problem, the composition of semantic web services, is introduced in the next chapter.

Chapter 3

Introduction to Semantic Web Services and Composition

Semantic Web Services (SWS) composition is a modern challenge for reasoning tools. Solving this problem requires to reason about different knowledges at both technical and abstract levels: services behaviour, functionalities and exchanged data. In this chapter we give an introduction to SWSs and the problem of composition. As research and notions in this domain evolve quickly, we settle on basic definitions for the elements of the semantic web that are involved in composition.
In the first part, we present SWSs. We discuss how their behaviour can be described and present workflows as a potential solution. We also present the nature of ontologies, which are the building blocks in the semantic web. In a second part, we describe and give a definition of the composition problem. In a third part, we survey the existing techniques for SWS composition, pointing out their main limitations and arguing the choice of configuration as a viable option.

3.1 Introduction to Semantic Web Services

3.1.1 Web Services

A *service* is a software agent providing a functionality. Up to now the web development has focused more on document exchange between humans than combined functionalities. The benefits of services interactions for knowledge sharing and business applications has become prominent with the success of the World Wide Web. Web-based services (WS) have developed associated standards such as SOAP or WSDL to provide a common communication interface.
SOAP (Simple Object Access Protocol) is a standard for the exchange of data between services using XML (eXtensible Markup Language). WSDL (Web Service Description Language) is the first standard developed for describing web services:

> WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-

> oriented information. WSDL is extensible to allow description of endpoints and their messages regardless of what message formats or network protocols are used to communicate.[1]

A central question in WS development is the ability to define interactions between them in order to achieve a new functionality. This implies the knowledge and advertisement of their behaviour.

3.1.2 Workflows and workflow patterns

Describing the behaviour of a WS requires the description of a complex process in the presence of asynchronous communications, interleaved parallel and sequential execution of tasks or sub-services, synchronization, etc. Such descriptions can easily be captured by *workflows*.

Definition 3.1.1 (Workflow) *A workflow defines the chaining, parallelisation and synchronization (the* control flow*) of atomic or composite activities which produce, transfer and consume data (*the data flow*).*

Workflows are usually modelled using graph related diagrams or boxed structures. There are several languages that have been developed to represent workflows based on different formalisms like (colored) Petri nets, Process algebra or BPEL (Business Process Execution Language).
In [109], Van der Aalst identifies 20 workflow patterns in order to compare different languages expressiveness. Patterns include behaviour like concurrency, synchronizing, alternatives, external choice, loops, etc. Several work on patterns [113, 88, 89] show that the languages based on colored Petri nets semantics like YAWL [108] or UML2 Activity Diagrams[2](UML2AD) offer the highest support of workflow patterns. We present in the following some of the workflow formalisms and languages.

Process algebra

Process algebra (also known as process calculus) is a collection of primitives and operators for describing processes. The interactions between processes are described using messages exchanged through *channels*. Algebraic laws are defined on operators which allow for equational reasoning on expressions. Operators usually include parallel and sequential composition as well as recursion and replication. Examples of process calculi include CCS(Calculus of Communicating Systems) [72], ACP(Algebra of Communicating Processes) or π-calculus [73].
Process algebras offer well-defined semantics and a set of operators useful for automated reasoning. However its box-language nature restricts the workflow modeling possibilities.

[1] Quoted from http://www.w3.org/TR/wsdl
[2] http://www.omg.org/technology/documents/formal/uml.htm

BPEL4WS

BPEL4WS is an executable modeling language based on XML. It is an extension of BPEL for describing interactions between WSs which itself is a combination of the languages WSFL and XLANG defined respectively by IBM and Microsoft. BPEL4WS is visually represented using the BPMN (Business Process Modeling Notation). BPEL4WS has a large tool support but suffers from the lack of formal foundations which makes difficult any automated reasoning on it. However several translations from different formalisms to BPEL4WS are available.

(colored) Petri nets

Petri nets were invented by Carl Adam Petri in 1962. Petri net is a modeling language which consists of places, transitions, and directed arcs between them. Places can contain *tokens*, and tokens can *fire* transitions. A fired transition will *consume* tokens from its input places and *produce* tokens in its output places. The formal foundations of Petri nets allow for computing properties such as reachability of states or liveness of a net (absence of dead-locks). Colored Petri nets [79] are an extension where tokens can have values such as types.

YAWL YAWL (Yet Another Workflow Language) [108] is a language based on (colored) Petri nets. It has been developed to directly support all workflow patterns. Although it cannot be criticized from the formal or expressive point of view, YAWL suffers from a poor acceptance and a lack of tool support.

UML2AD Activity Diagrams have evolved from UML 1.5 state-machines to token flow semantics which now allows them to modularly support workflow patterns. If the token flow of UML2AD provide means to freely model and document the behaviour of web services, its mapping to Petri nets is far from obvious, because of doubts concerning the exact semantics of some of the constructs. However a number of researches [12, 102] have defined conditions on UML2AD subsets for such a mapping. As part of an ongoing effort from the Object Management Group[3] to standardize software design, UML2AD has a wide acceptance among engineers and industrials as well as a large tool support. Figure 3.1 shows a simple activity diagram which could model a web service behaviour. We now give an overview of activity diagrams since we will use them in the next Chapter. This short presentation can also serve as a basic explanation of Petri nets and token-flow mechanisms.

Diagrams, nodes and edges The main constituents of activity diagrams are *activity nodes*, arranged in *activity groups*. These nodes are connected by directed *activity edges*: *object flows* and *control flows*. All edges have the ability to transport *tokens* from source to destination. Object flows can hold data tokens whereas control flows, via control

[3] http://www.omg.org/technology/documents/formal/uml.htm

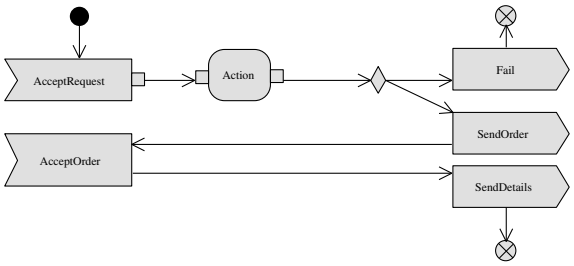


Figure 3.1: An example of a UML2 activity diagram

tokens, imply that a node execution must precede another.
There are three types of nodes. *Object nodes* specify which data goes through object flows. *Action nodes* refer to a local process being executed and have special object nodes called *pins* attached to their input or output object flows. *Control nodes* allow to define parallelisation and synchronization of tasks as well as alternative paths.
As a general rule, each type defined by an object node reached by a data token must be more general than (or equivalent to) its incoming edge source type.
An important element in the context of SWS is the *Event* node. *AcceptEvents* and *SendEvents* denote that the workflow is respectively waiting for or sending a message to an external agent.

Token flow and traverse-to-completion semantics Activity nodes do not contain tokens, instead they consume and create them. In general, the finishing of a node causes tokens to be created in every outgoing flow edge and a node can only be executed if all its incoming flow edges and input pins yield at least one token each. A noticeable exception to this rule is the case of *decision* and *merge* control nodes. Those nodes allow for alternative paths in the workflow. A decision node will output a token on one of its outgoing edges depending on a given condition (the *guard*). A merge node will be executed whenever one of its incoming edges has a token.
Traverse-to-completion means that tokens will only leave their current position and move on to another token-consuming node if the whole path to the destination node is executable. This is especially interesting for control nodes, as the whole path (though composed of numerous elements) acts like a single Petri-net transition.

Interruptible Activity Regions This construct allows to model the "external choice" workflow pattern. This pattern expresses the fact that the process is waiting for an external entity to make a choice, on which depends the outputs of the process. The UML2AD construct is an activity group containing special activity edges called *interrupting edges*. The execution semantic is that whenever a token traverses an interrupting edge, all tokens present in the associated interruptible group are removed. For instance, in Figure 3.1, events waiting for "fail" and "SendOrder" should each have an outgoing interrupting edge thus cancelling the other when the message is received.

3.1.3 Semantic Web Services

One idea about WSs is to automatically process or re-use them in Service Oriented Architectures (SOA). However, the data manipulated by these services is usually of different nature and does not necessarily yield a shared understanding. For instance, a rental service may talk about "cars" whereas a parking service talks about "vehicles". As said previously, the behaviour of a WS is also described in many formalisms. Another important aspect is that SWSs owners often do not want to advertise their internal processes, but only the least necessary part required to allow clients to enter into valid interaction.
Ontologies have been introduced as a potential solution. An ontology can be described as a *shared conceptualization of a domain*. Common ontologies would enable easier communications between services.
Two types of ontologies can be distinguished in the domain of semantic web services: *data ontologies* describe the data in the exchanged messages whereas *service descriptions* describe what the service achieves and how. The addition of semantic technologies to web services aims to allow machines to automatically reason about data and processes. We can use the following definition:

Definition 3.1.2 (Semantic Web Services) *SWSs are software agents which publish their functional and behavioural interface so as to allow reasoners to help discover, invoke, compose or adapt them.*

Data ontologies

Definition 3.1.3 (Ontology) *An ontology is a data model for a knowledge domain representing a set of concepts through classes, attributes and relationships.*

Several formalisms have been proposed to describe ontologies. Semantic web research has built on top of existing technologies such as XML[4] as a data organization format, and later RDF[5] which opened the path by allowing simple semantics to be attached to XML. However, the complexity of knowledge requires a higher degree of expressivity but the underlying logic should limit the computational complexity in reasoning tasks. In particular, the problem of termination (decidability) is essential for efficient reasoners.

OWL Description Logics [13] allow to express taxonomies and partonomies very precisely. A description logic ontology consists of concepts, roles and individuals. Concepts represent sets of real-world entities having common characteristics, individuals are instances of the concepts, and roles represent connections between individuals. OIL and DAML+OIL[6] are the first languages which tried to use description logics in the field

[4] http://www.w3.org/XML/
[5] http://www.w3.org/RDF/
[6] http://www.w3.org/TR/daml+oil-reference

of the semantic web, later giving birth to OWL[7] (for Ontology Web Language). Three versions of OWL currently exist:

- OWL-Lite has been designed to allow for easy implementation of reasoning systems. It supports classification hierarchy and simple constraints. As an example of its limitations, it only supports cardinality values of 0 or 1. OWL-Lite is a notational variant of the SHIF(D) description logic [52],
- OWL-DL increases the expressivity of OWL-Lite without loosing decidability. It is a notational variant of the SHOIN(D) description logic [52],
- OWL-Full is a logic compatible with the syntactic freedom of RDF, but without computational guarantees. It is not known whether OWL-Full is decidable or not. A class in OWL-Full can be treated simultaneously as a collection of individuals and as an individual, making it close to second-order logic.

WSML Another direction has been taken with the WSML[8] (for Web Service Modeling Language), developed recently through European projects. WSML also supports several versions:

- WSML-Core is defined by the intersection of description logic and horn logic,
- WSML-FLIGHT and WSML-RULE are extensions in the direction of logic programming. WSML-FLIGHT is based on a logic programming variant of Frame Logics [59]. Frame Logic has been designed to manipulate object-oriented logical databases. It does not provide non-monotonic features, such as default inheritance, type checking or well-founded negation,
- WSML-DL is an extension of WSML-CORE which captures the description logic SHIQ(D). It is not fully specified yet,
- WSML-FULL intends to unify WSML-DL and WSML-RULE. It is still an open research.

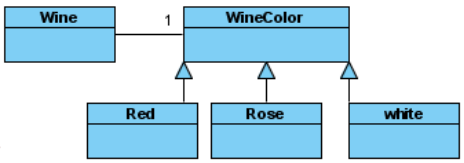
Figure 3.2 shows a fragment of a wine ontology and its description in OWL and WSML.
In the context of SWSs, instances of *concepts* taken from an ontology describe the types of *messages* sent and received by web services.
As services may not share the same ontology for a similar knowledge, the semantic web introduces *mediators* as transformations between concepts. Mediation can be defined manually or automatically through ontology reasoning. One example is the transformation of money amounts from one currency to another.

[7] http://www.w3.org/TR/owl-features/
[8] http://www.wsmo.org/wsml/

OWL

```
<owl:Class rdf:ID="Wine"/>
<owl:Class rdf:ID="White">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="WineColor"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="wineColor">
  <rdfs:domain rdf:resource="#Wine"/>
  <rdfs:range rdf:resource="#WineColor"/>
</owl:ObjectProperty>
<owl:Restriction>
    <owl:onProperty rdf:resource="#wineColor"/>
    <owl:cardinality rdf:datatype="&xsd;nonNegativeInteger">1</owl:cardinality>
 </owl:Restriction>
```

UML



WSML

```
<concept name="http://www.example.org#Wine">
  <attribute name="http://www.example.org#wineColor" type="inferring">
    <range>http://www.example.org#WineColor</range>
    <minCardinality>1</minCardinality>
    <maxCardinality>1</maxCardinality>
  </attribute>
</concept>
<concept name="http://www.example.org#WineColor"/>
<concept name="http://www.example.org#Red">
  <superConcept>http://www.example.org#WineColor</superConcept>
</concept>
```

Figure 3.2: A fragment of a wine ontology in UML, OWL/XML and WSML/XML

Service descriptions

It is claimed that there are two separate characteristics of web services that must be captured to form SWSs: the *functional* and the *behavioural*. The functional part describes what the services achieve in terms of *input and output messages*, and may also define *preconditions and effects*. The behavioural part describes how the services achieve it. An approach on behavioural descriptions is that two related descriptions coexist. From the client's point of view, it should describe the message exchange patterns so as to be able to engage communication. This description, which can hide most of its internal process, is called the *choreography*. It is believed to be the advertised part of a service. On the other hand, the *orchestration* describes how the service may use external services and internal elements to achieve its functionality.

Definition 3.1.4 (Choreography) *A choreography is the behavioural description of how a client can communicate with a web service to achieve its functionalities.*

Definition 3.1.5 (Orchestration) *An orchestration is the behavioural description of how a composite web service uses external services to achieve its functionalities.*

As a technical description of web services, WSDL does not provide any means to semanticly enrich them with behavioural and capability descriptions. The recently proposed

standard SA-WSDL allows to reference such descriptions. In the past years, two main representations for SWSs descriptions have emerged: OWL-S[9] and WSMO[10].

OWL-S In OWL-S, the *service profile* contains the functional characteristics of the SWS, while the *process model* defines the behaviour. The *service grounding* provides the details for transport protocols such as WSDL. Figure 3.3 shows the 3-stacks service ontology. The OWL-S process model is an algebra of workflow forms, called processes, where the atomic processes are grounded to operations on web services. This process model offers interesting features among which the hierarchical decomposition via control flow. On the other hand, the process model does not make an explicit distinction between choreography and orchestration and also suffers from serious expressive power limitations. For instance, the external (or deferred) choice workflow pattern is not supported by the process model. This is obviously an important feature in the context of semantic web services interactions.

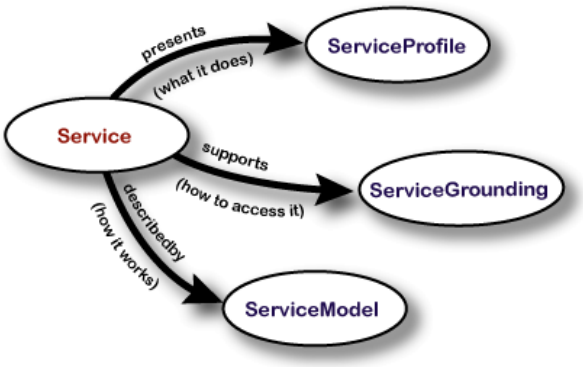


Figure 3.3: OWL-S service ontology

WSMO In WSMO, the functional definition is in the *capability*, and the behavioural part is in the *interface* made up of choreography and orchestration descriptions. The service ontology also allows to define non-functional properties for information such as service owner, quality-of-service or trust. Figure 3.4 is an overview of WSMO service ontology. The WSMO meta-model is expressed in the OMG's meta-object facility and then given a grammar to form the Web Services Modeling Language (WSML), in both a human-readable and an XML syntax, over which reasoning is defined. WSMO also defines mediators and goals (user requests) as top-level entities.

[9] http://www.w3.org/Submission/OWL-S/
[10] http://www.wsmo.org/

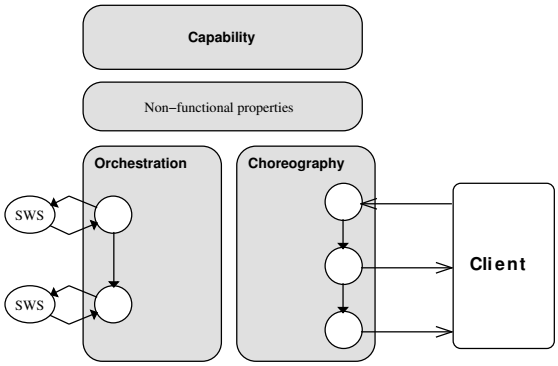


Figure 3.4: WSMO service ontology

Goals and Discovery

A user looking for a service will express requirements on what he expects to be achieved: a *goal*. Goals can be expressed as required capabilities, in the same language as a SWS's functionality. The user may also specify how he wants to interact with the service, in this case the goal will contain a client choreography. The process of matching a goal with one or multiple SWSs is called *discovery*.
UDDI (Universal Description, Discovery and Integration) was the first registry allowing to publish WS descriptions on the internet. However it acts as a simple list and does not provide any means to reason about requests besides syntactical matching.
Indeed, the development of functional semantic descriptions such as capabilities now offer the possibility to retrieve SWS based on different types of semantic matchmaking. It is usually based on inputs and outputs concepts comparison between the user's goal and the service capability. We can list the main types of matchmaking between concepts:

- *equivalence* for strictly equivalent concepts,
- *disjoint* for strictly different concepts,
- *overlap* (intersection) for concepts having partly shared domains,
- *subset* for a concept included in another,
- *superset* for a concept superseding another.

Several reasoners [18, 81] have been developed for discovery engines.

3.2 Introduction to SWS Composition

Composition is often defined as the "act of combining and coordinating a set of Semantic Web Services". The result of such a composition can itself be a SWS. Under such a definition, composition naturally refers to the process involved in computing such a combination. We have seen in the previous section that the coordination of different SWSs can be described at the process-level by an "orchestration". We thus consider issues that arise when addressing the task of automatically or manually designing an orchestration from a set of available SWSs.

3.2.1 Composition inputs

A composition tool requires different inputs:

- *User request*: a user, would it be a client or an engineer designing a new SWS, will make requirements on the expected result. An end-user may not know that composition is necessary for his goal to be fulfilled, it can be an invisible process fired by the non-existence of a matching SWS. In this case we will talk about *on-the-fly* composition. The required functionalities will thus be expressed with the classical notion of goal introduced previously.
  However, in the case of an engineer creating his own SWS (we talk about *design-time* composition), the request will probably be more precise: using external SWSs having specific properties such as geographical location, ordering of processes, definition of alternative paths, etc. As this information is not contained in classical goals, we will call these problem-specific requests *composition goals*.

- *Available services*: a composer needs a set of web services it can use in the orchestration. This set can be obtained through discovery if a set of capabilities/goals is provided or computed. Discovery requests can be interleaved with composition or executed once-for-all. An automatic computation of required capabilities can be seen as *goal decomposition*.

- *Data ontologies*: the set of ontologies and concepts used by a SWS is usually referenced in its description, thus available to the composer.

3.2.2 Composition objectives

The result of a composer can take different forms depending on the chosen level of abstraction.
On the one hand, one can consider that it only describes the required (atomic) capabilities and some knowledge about the implied interactions, leaving apart computing tasks like discovery, process description or data mediation. In this case we will talk about

capability-level or *goal-level* composition. This result is sometimes called an "orchestration of goals" in the literature. Such approaches can hardly be used for on-the-fly composition, require an additional and compatible tool for finalizing the composition, and may describe unrealisable orchestrations. However they allow for late-selection of participant SWSs.
On the other hand, a complete composition is the production of an orchestration tied to specific SWS choreographies. We will talk about *process-level* composition. It offers the ability to directly execute the composition result.
In the purpose of publishing its result as a new service, a composer may also describe the composite SWS capability and choreography.

3.2.3 Problem Definition

Definition 3.2.1 (Goal-level SWS Composition problem) *Let R be a request, G be a library of available goals or capabilities and Onto be a set of related ontologies. A goal-level composer produces an abstract description of a composite web service which may achieve the functionalities required in R. The description is made of a set of goals $g \in G$ and a set of abstract requirements for a potential orchestration.*

Definition 3.2.2 (Process-level SWS Composition problem) *Let R be a request, C be a library of available choreographies and Onto be a set of related ontologies. A process-level composer produces a description of a composite web service which may achieve the functionalities required in R. The description is an orchestration O.*

3.3 SWS composition: practical approaches

There are numerous works in the field of SWS composition, however many introduce languages and formalisms for manual composition, or verification and diagnosis of orchestrations. In this Section we only list known scientific approaches to SWS (semi-)automatic composition and present for each of them:

- the core features and expressive power of the formalism,
- the level of composition addressed (supported elements and issues),
- the implementing tools and conducted experiments

3.3.1 Situation calculus

In the Situation Calculus (SC) [62], originally proposed by McCarthy and Hayes in 1969, first order logic is applied to the description of world states (or *situations*) and side effect actions. The Situation Calculus allows one to reason about valid moves, reachable situations and raises issues of strong concern in AI planning such as the frame problem.

The essential idea in SC is to replace predicates by *fluents*: predicates indexed with situations. SC introduces special predicates like *poss* (can an action be performed in a given situation?), *holds* (is a formula true in a given situation) and *do* (perform an action provided its preconditions are met). Preconditions are naturally stated using poss.

Golog

Golog is a high level specification language built on top of the situation calculus, with knowledge and sensing actions. Golog introduces a number of extra logical constructs for assembling primitive actions, defined in SC, so as to form complex actions that may be viewed as programs. Existing Golog interpreters are Prolog based.
In [69], the Golog extension ConGolog (previously introduced for "concurrent" Golog) is shown to be suitable for Web Service composition with two extensions. To circumvent the fact that the "sequence" Golog construct is static, and allows for no insertion of actions, [69] introduces an extraneous "Order" construct, that allows the dynamic insertion of an action so as to fulfil preconditions.
However in [69], web services are not automatically composed. Instead they create manually generic procedures that can be reused by users with their own preferences and constraints. Furthermore, the generic procedures do not support web services having non-atomic choreographies, i.e the communication pattern is a one-shot "send and receive".

3.3.2 Logic Programming

The work in [95] illustrates the possibility of using Prolog to interactively generate Web Service compositions based on their semantic descriptions (originally in WSDL). This approach emphasizes the possibility of viewing Web Service composition as a recursive process, and advocates the use of well known AI techniques in the field. Specifically, the possibility offered by Prolog for an end user to interactively control a composition program is interesting. Such a system can efficiently exploit semantic conditions, and also can explore an entire combinatorial search space. The authors of [95] claim their system gives a straightforward account of WSDL specifications. The prototype implementation performs on the basis of previously discovered Web Services, hence does not contribute to the discovery process. The limitations of using (standard) logic programming are as usual: because of the Horn clause sublanguage, direct support for some logical formulation turns to be difficult, such as those involving disjunctions and existentials.
Such a system can, however, non-deterministically explore the complete search space of possible compositions and account for many constraints.

3.3.3 Type matching

The cost of Web Service discovery queries, together with expected non availability of advertised choreographies and capabilities offering exact matches induces a promising way of composing Web Services on the basis of partial matches of their input/output

data types.
Type description/reasoning itself can be performed in one or another language, from description logics to mixins based languages. The composition algorithms themselves range from ad hoc recursive procedures to standard logic programming forward/backward chaining. Type matching based composition hence deserves a separate section in this state of the art.
A "mixin" is an object oriented concept whereby the fact that an object matches a given programming interface, e.g., allowed to call a given method is resolved at run time. In contrast, Java or C++ interfaces are resolved at compile time.
The work in [19] details an algorithm for Web Service composition with partial type matches, and shows that such an approach significantly improves the number of successful compositions. Interestingly enough, the composition algorithm interleaves the composition task with the Web Service discovery, which addresses several practical problems:
- The discovery process should be as efficient as possible, and type inference can in many cases be made very efficient ([16] uses numeral representations of types),
- The number of discovery queries should be limited as much as possible, due to the significant overload induced by such remote complex queries performed on potentially huge databases.
Partial type matching approaches to Web Service composition reinforce the intuition that an essential issue regarding this problem has to do with the type of data exchanged between peer services (this is also a leading intuition in problem solving methods). Partially matching inputs/outputs can be adapted, reorganized, grouped together, so as to build a working system from disparate and literally incompatible elements. In that sense, reasoning about type compatibilities appears as a central requirement to Web Service composition, prior to more advanced forms of reasoning.

3.3.4 Modal Action Logic

Multi agent systems share a lot in common with the semantic web. Formal agent conversation languages may be applied in some cases to workflow composition problems because they also deal with protocols. The full interaction of several agents can be perceived as a complex choreography in a SWS sense.
As an example of the proximity of the two fields, in [6] Web Services are viewed as actions, either simple or complex, characterized by preconditions and effects. Also, interaction is interpreted as the effect of communicative action execution, so that it can be reasoned about. The formal language used is a modal action and belief logic DyLog. The language allows an agent to reason about the interactions that it is going to enact for proving if there is a possible execution of the protocol, after which a set of beliefs of interest (or goal) will be true in the agent mental state.

3.3.5 Linear logic

Linear logic [42] is an extension of classical logic to model a notion of evolving state by keeping track of resources. Other resource aware logics were developed before, but LL has attracted a lot of research attention. Specifically, LL has well defined semantics and provers are available.
The paper [86] proposes an application of LL to Web Service composition. The authors claim that the WSDL presentation of a Web Service can be automatically translated to a set of LL axioms. Then, they use a prover for the multiplicative propositional fragment of LL to infer the composition of a Web Service. The target Web Service is described as a sequent in LL, to be proved by the proof system. The context is a restrictive case (the "core" Web Service is known, but not some of its value added sub services). Being complete, the system can generate all possible compositions. Each composition, available as a sequent proof, can be translated to a WS-BPEL workflow.
This original approach still faces several limitations, acknowledged by the authors themselves ("the full automation of the composition process is a difficult problem"), like the fact that the logic used is "only" propositional.

3.3.6 Problem solving methods

The problem solving method (PSM) [9] describes the foundational ontologies for the UPML language [32]. As such, PSM is not a formal system, but forms a model of processes that can be used to compose semanticly described Web Services. The work in [43] describes a possible framework for using PSM in that objective.
The essence of PSM is to provide a distinction between methods and their abstraction called tasks, and to focus on the inputs and outputs of tasks and methods, described using ontology based pre/post conditions. This approach treats workflows as secondary relative to the logical conditions necessarily matched by viable processes. For instance, the preconditions satisfied by composite tasks must match the preconditions of their starting subtasks. This viewpoint is essential to Web Service composition, where determining whether Web Services are I/O compatible is necessary even before testing that their choreographies are compatible.
In [25], a viable connection between PSM and OWL-S is presented.
The intuitions underlying the PSM model can be related with practical experimentations conducted with the Ariadne mediator system, as documented in [41]. This work shows how input/output requirements for Web Services can be exploited using a simple forward chaining algorithm, according to the following idea: the user feeds in the system with a description of the data they can provide, plus a description of the data they request from the (dynamically composed) system. The composition algorithm recursively loops adding Web Services that produce some of the desired information. Each new required input not currently available is further treated as desired. The system stops when a set of Web Services has been constructed that produces the expected output from the available initial input. Although [41] does not account for the compatibility of

Web Services choreographies, the proposed working system validates several important intuitions regarding Web Service composition.

3.3.7 Process algebra

As OWL-S process models are based on process algebra, they constitute a good choice for composers willing to exploit this emerging SWS formalism.
The use of process algebraic languages (like CCS [72] or Pi Calculus [73], but also CSP and LOTOS) which originate from the rich field of concurrent programming and systems has been advocated for Web Service composition. The constructs of several process languages for the semantic web have been "a posteriori" formally grounded on process algebras. The work in [8] details a possible formal account of WSCI using CCS, and points to accurate bibliography in the field. Model checking methods can be used to automate or assist the composition/validation process.
CCS [33] is the simplest process algebra. A CCS grammar is defined using "processes", "channels", data items, and sequences of values. A process can be prefixed by an atomic action, or composed with other processes, either in parallel 'jj' or by means of the choice '+' operator. Atomic actions are either the internal (or silent) action "T", input actions (a message "x" is received from a channel "a") or output actions (a message is sent through a channel). The operational semantics of CCS is defined by a transition system where standard rules model parallel and choice operators, and synchronization is produced by the parallel composition of two complementary actions. In spite of its simplicity, CCS presents a high expressive power, capable of capturing WSCI as illustrated in [15].
Ad hoc and more complex process algebras than CCS can be designed to formally define core subsets of process languages. The work in [110] illustrates this, giving precise formal description of the semantics of a core Composition subset of BPEL. The originality of the process algebra in [110] is that besides standard process algebra constructs (e.g., choice, sequence) it provides notations for iterative cycles and variable assignment (that stem from standard programming languages).
The Cashew [77] language gives compositional semantics to OWL-S, together with explicit support for several workflow patterns, including external vs. internal choice.

3.3.8 Planning

SWS composition can obviously be viewed as a planning problem: we look for a plan of actions (i.e., Web Services) which guarantees that the target objective will be reached. A planning problem can be described as a five-tuple $< S, s0, G, A, T >$ where S is the set of all possible states of the world, s0 denotes the initial state of the planner, G denotes the set of goal states the planning system should attempt to reach, A is the set of (ground) actions the planner can perform in attempting to reach a goal state, and the transition relation $T : SxA \rightarrow S$ defines the semantics of each action by describing the state (or set of possible states if the operation is non-deterministic) that results when a

particular action is executed in a given world state.
Planning offers first-order logic expressive power together with several solving algorithms. Therefore it is not surprising that many SWS composition researches are based on planification.

SWORD One of the tentatives is the library for interactive Web Service composition SWORD [84] where the plans are generated using a rule based forward chaining algorithm. SWORD is based on non-semantic technologies (WSDL). Therefore the support of ontologies is limited to the equivalence of concepts and no account is taken for choreographies of composed web service (each service is an atomic operation). Furthermore, the question of complex requests is not studied.

SHOP2 Hierarchical Task Network (HTN) planning is an AI planning methodology that creates plans by task decomposition. This is a process in which the planning system decomposes tasks into smaller and smaller subtasks, until primitive tasks are found that can be performed directly. SHOP2 is a domain-independent HTN planning system. A planning problem in SHOP2 is a triple (S, T, D), where S is initial state, T is a goal task list, and D is a domain description. SHOP2 will return a plan P = (p1 p2 ... pn), a sequence of instantiated operators that will achieve T from S in D. SHOP2's knowledge base contains operators and methods. Each operator is a description of what needs to be done to accomplish some primitive task, and each method tells how to decompose some compound task into partially ordered subtasks. An application of SHOP2 to SWS composition is presented in [96]. The authors give a translation of DAML-S processes to SHOP2 methods and domains. The presented composer handles most of DAML-S but there is no support for concurrency processes which limits the potential web services choreographies and generated orchestrations. The translation of complex requests to SHOP2 planning requirements, the combination with discovery or the reasoning on concepts are not studied either.

GraphPlan SAT based planning is largely studied, because of the possibility to exploit efficient heuristics and cuts, and also thanks to recent improvements in SAT solving alone. [114] extends GraphPlan (a SAT planning algorithm) to automatically generate the control flow of a web process. The proposed extension also handles compatibility between input and output messages, as well as data mediation through middleware or externalized web services. Their composer is able to produce an executable BPEL process from SWS annotated using SA-WSDL. Unfortunately the expressivity of web services behaviour is restricted to the sequence, loop (in a post-process step) and AND-split workflow patterns with composite web services being made of atomic operations without any conditional outputs. The request language is limited to an initial state and a goal state, and no account is taken of concepts reasoning.

State transition systems and planning via model checking In [106, 83], an efficient approach to process-level composition is proposed. Candidate web services choreographies are first translated into a set $\{\sigma_{W1}, \ldots, \sigma_{Wn}\}$ of state transition systems (STS). STS describe dynamic systems that can be in one of their possible states and can evolve to new states as a result of performing some actions (message exchanging but also "invisible" actions for internal changes). Then they compute the parallel combination $\sigma_{||}$ of the systems. The set of requirements is expressed using the *EAGLE* language [10]. The composite service is finally obtained with a plan synthesis technique (planning as model checking) that has been adapted to take into account the partial observability due to "invisible" transitions and non-deterministic behaviours.
The proposed framework is able to deal with and produce SWSs described using BPEL4WS or OWL-S. In the case of OWL-S, they propose an extension in order to deal, for instance, with the external choice pattern. Their tool also makes a difference between *on-the-fly* composition and *once-for-all* composition, where the created compositions allow for complex interactions and negotiations with the user. Finally, the *EAGLE* language allows to state complex requirements on both the control-flow and the data-flow. The approach is, to the best of our knowledge, the most efficient and elaborated method for SWS composition. Moreover, unlike other parroaches, they present experimental results with composition times hence allowing for comparisons. The main limitations are the inability to do any type of data reasoning, as well as its process-level restricted nature which does not allow for discovery interaction.

3.3.9 Comparison of existing approaches

Table 3.1 gives a comparative summary of existing approaches. We consider the following features:

- (CompLevel) the level of composition achieved i.e goal-level and process-level,
- (NAC) the support for non-atomic choreographies i.e not only one-shot services,
- (WP) the support for three sets of workflow patterns: (a) basic control flow (sequence,concurrency); (b) basic data flow (internal choice,external choice, synchronizing merge); (c) structural patterns (loops, multiple instances),
- (DR) the support of data reasoning i.e reasoning besides strict equivalence of concepts,
- (SWS) the support of existing SWS formalisms i.e DAML-S, OWL-S, WSMO, BPEL4WS,
- (Req) the availability of a language for complex requests i.e able to express requirements other than input and output messages,
- (Disc) the interaction with a discovery tool,
- (Tool) the existence of a tool for automatic composition or at least specifications.

	CompLevel		NAC	WP			DR	SWS	Req.	Disc.	Tool
	goal	process		(a)	(b)	(c)					
[69] (Golog)	-	-	-	+	+	+	-	+/-	+	-	+
[43] (PSM)	-	+	+/-	+	+	+	-	+	-	-	-
[41] (Ariadne)	+	-	-	-	-	-	+/-	-	-	+	+
[6] (Modal)	-	+	+	+	+	-	-	+	-	-	-
[95] (LP)	-	-	-	-	-	-	+/-	+	+/-	-	+
[19] (type-matching)	+	+	-	+/-	-	-	+	-	-	+	+
[77] (Cashew)	-	+	-	+	+	+	-	+	-	-	-
[96] (SHOP2)	-	+	-	+/-	+	+	-	+	-	-	+
[114] (GraphPlan)	-	+	-	+	-	+/-	-	+	-	-	+
[84] (SWORD)	+	-	-	+/-	+/-	-	-	-	-	-	+
[83] (STS)	-	+	+	+	+	+/-	-	+	+	-	+

Table 3.1: SWS Composition approaches comparison (+) = supported, (-) = not supported, (+/-) = partially supported

3.4 Conclusion

The semantic enrichment of web services descriptions aims at allowing automatic reasoning on them. The ability to create composite services is naturally a key feature for the semantic web. We presented in this Chapter a tentative state of the art of current research in this quickly evolving domain, and tried to define the core elements involved in composition. We also identified different levels of composition, reasoning either at the level of capabilities or at the level of processes, and highlighted the features required for a practical use of composition.
We listed several existing approaches, and pointed out their limitations. Configuration emerges as a potential option for different reasons. Data ontologies, which are the building blocks of the semantic web, are expressed using description logic. Constrained object models are therefore a potential representation choice over which direct reasoning is possible. We also showed how services behaviour can be well described using workflow languages. A workflow description being a set of interconnected objects, one can easily imagine its representation as a configuration model. Finally we advocated that composition calls for first-order logic formalisms. Configuration thus appears as a viable choice. The next Chapter presents a composition approach based on configuration from constrained object models.

Chapter 4

Configuration-based SWS Composition

SWS composition is an opportunity to apply COMs and configuration to a domain where complex and highly symbolic reasoning is required, therefore contributing to the usefulness of configuration in general artificial intelligence problems. This research aims at:

- providing a robust and viable solution to the challenge of SWS composition,
- demonstrating the expressive power and advantages of configuration in a domain originally interested in syntactic approaches, theorem proving and process algebra,
- extending the application domain of configuration to abstract knowledge and symbolic reasoning,
- pointing out the issues arising when using configuration in modern artificial intelligence problems.

We propose a complete and reproducible description of a theoretical and experimental framework, based upon the use of COMs and configuration, to address composition. We do not intend to give formal foundations or a general theory to the problem of SWS composition, but rather to describe a working configuration-based solution within clearly defined limits. It is experimentally validated by its integration in a full SWS framework developed through the DIP European project[1] which includes SWS descriptions, discovery, mediation and execution. This work has led to several publications [3, 4, 50]. It has also been presented in the European project's deliverables [1, 2], and influenced standards developed in deliverables [11, 37, 61].
We first give an overview of the whole composition approach. We then define in details

[1] DIP European project webpage: http://dip.semanticweb.org

the different steps involved in the process. We describe our implementation of the composer as well as its integration and contributions to the project's developed framework. Finally we provide experimental results and analyse the strengths and weaknesses of the approach.

4.1 Overview of the two-level configuration-based approach for SWS composition

As introduced in the previous Chapter, SWS composition can be realized either at the goal-level or at the process-level. In the following, we propose to apply configuration at both levels: a configurable request language is used to drive the configuration of workflows. We present an overview of this two-level process.

Process-level composition

A process-level composition should produce an orchestration out of available SWS choreographies. We consider the choreographies and orchestrations documented as workflows. We define a language, based on token-flow semantics, which is able to express both choreographies and orchestrations. The language is an adapted subset of the familiar UML2 activity diagrams (UML2AD). This choice is further explained in Subsection 4.2.1.

Configuring workflows We define a COM which captures workflows expressed in this language. The request for a composition is made of a set of user required outputs and available inputs messages. The object model and its constraints then ensure that the configurator creates a composite workflow achieving user objectives. This configured workflow contains the required external SWS choreographies as well as internal workflow elements.

Configuring syntactically valid workflows A set of constraints in the COM ensure that the composite workflows are syntactically valid. Beside the workflow language restrictions, the leading idea is that a valid workflow has all necessary inputs fulfilled by some adequate output based on their carried message types (in our context, concepts from ontologies).

Configuring useful workflows A composite workflow should guaranty that at least one possible execution path leads to all user objectives. In order to ensure this, we add a boolean attribute to all workflow atomic elements. If this attribute (called *active*) is true, then the element is expected to be activated during execution. A set of constraints in the COM control the propagation of this activity marker with respect to the execution semantics of the language. As the user objectives are set active at the beginning, the composite workflow has at least one active execution path to them.

Obtaining the SWS descriptions from the composite workflow A composite SWS is described with its capability, choreography and orchestration. Isolating the orchestration cannot be straightforward since the consumed choreographies contain workflow execution information. We propose a post-workflow composition process so as to compute these descriptions. The related issues and the procedure are described in Section 4.4.
The whole process-level composition process is illustrated in Figure 4.1.

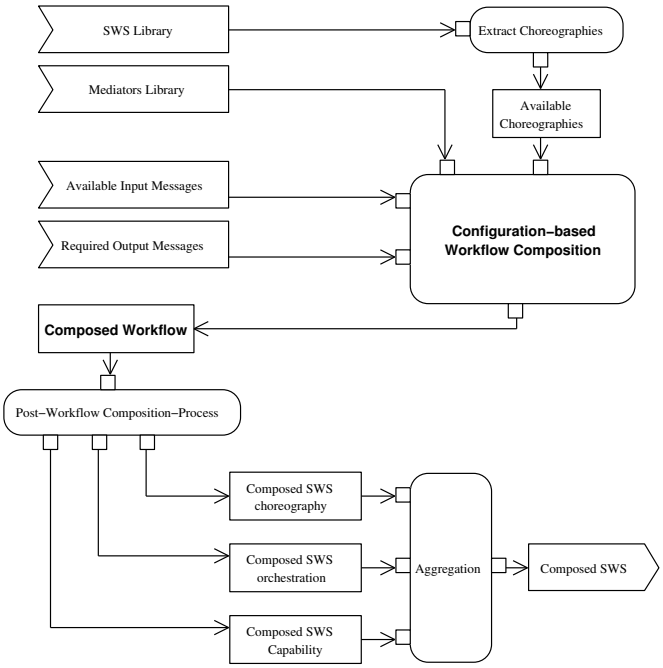


Figure 4.1: SWS workflow composition process

Goal-level configurable requests: composition goals

We define at the goal-level a composition request language, called composition goals, allowing to specify additional requirements on the desired orchestration. These are taken into account through a modular translation to the workflow configurator elements and constraints.

Interaction with discovery Composition goals allow to specify which atomic goals are required. Discovery can be applied on those to create a library of useful SWS for composition. This process significantly limits the number of choreographies fed into the configurator.

Configuring requests Another originality of our approach is that composition requests are themselves configurable because we describe them as a COM. This allows for automatic, semi-automatic or assisted user definition of the request, which we presented in the previous Chapter as goal-level composition.
The whole two-level composition process is illustrated in Figure 4.2. An example applied to a real-world scenario can be visualized with Figure 4.3 (a composition request), Figure 4.4 (a computed composition goal), and Figure 4.5 (a computed orchestration). This NMPC-bundle (Network Modem PC) scenario, developed with a use-case partner of the European project DIP, aims at composing a service able to sell a PC, a modem and a network connection to a user. The notations used in these Figures will be introduced in the corresponding Sections. However one can already notice that in Figure 4.4, a set of atomic goals has been computed (pictured as white rounded corner rectangles). In Figure 4.5, the corresponding set of consumed services participates to the orchestration.

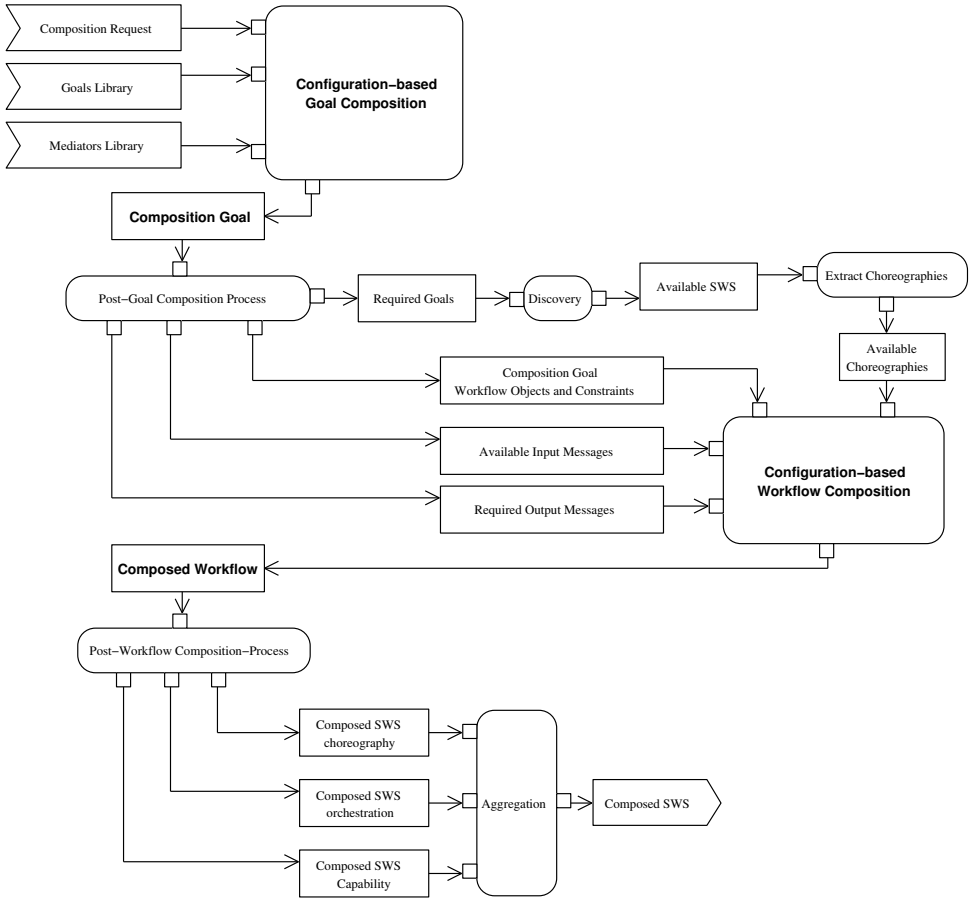


Figure 4.2: SWS composition process

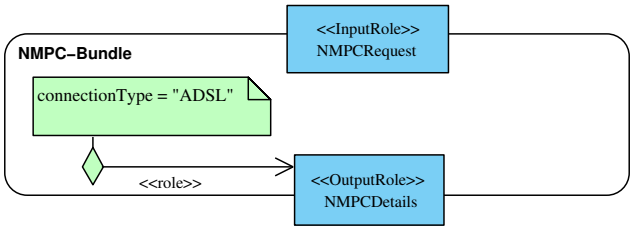


Figure 4.3: A composition request for the NMPC-bundle scenario

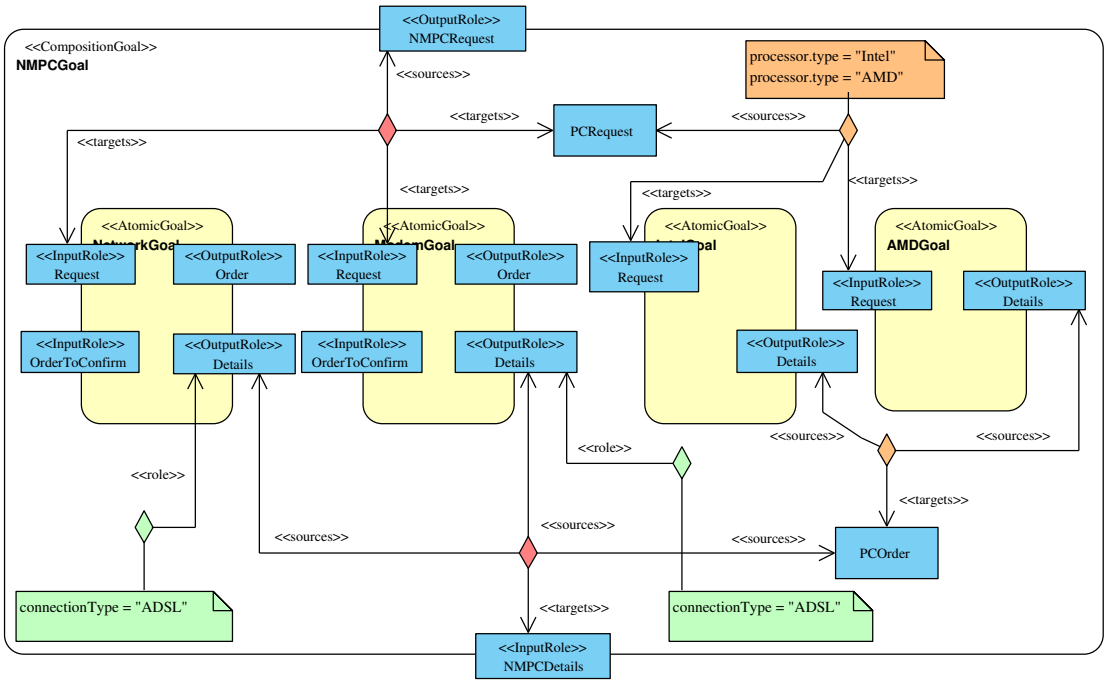


Figure 4.4: A computed composition goal for the NMPC-bundle scenario request

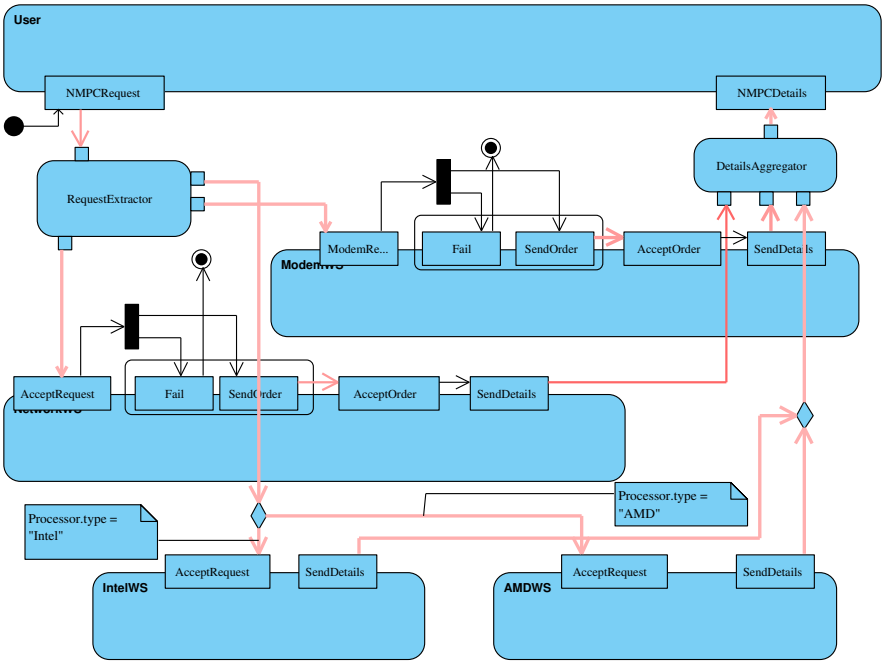


Figure 4.5: A computed orchestration for the NMPC-bundle scenario composition goal (bold edges represent object flows)

4.2 Process-level composition

Following the composition process introduced in Section 4.1, we present the process-level composition step. We first define the workflow language which expresses SWS orchestrations and choreographies through a complete COM, allowing for workflow configuration. We then describe the computation process.

4.2.1 A common workflow language for the specification of SWS orchestrations and choreographies

In the previous Chapter we argued that workflow nets (i.e. extensions and variants of Petri nets) are good candidate choice for SWSs behavioural descriptions. We also advocated that UML2AD, which are based on Petri nets and known to a wide audience, modularly conform to the workflow patterns [113, 88, 89]. However we also pointed out that parts of the UML2AD specifications raise issues on the execution semantics. We therefore isolated and adapted a subset of UML2AD, called AD-S, with well-defined semantics. The subset is able to express both SWS orchestrations and choreographies.

AD-S: a comprehensive subset of UML2AD

We gave an overview of UML2AD in the previous Chapter. The AD-S subset only includes constructs and features with well-defined execution semantics. Besides the limited number of allowed constructs, the following is a list of our subset's original properties and restrictions with respect to the UML2AD specifications:

- *duplicate edges* are not allowed. A duplicate edge refers to the fact that two edges are going out of the same pin. The specified behaviour called *token competition* does not allow for a deterministic computation of which edge will receive the token,
- we restrict the allowed diagrams to *acyclic diagrams*. In particular, this prevents some self-blocking situations.

Researches [102, 111] have defined restrictions on UML2AD in order to define a mapping to coloured Petri nets. Although these restrictions are not necessary in our COM, complying ensures formal foundations to our language.

- *1-safe-nets* is a restriction on the number of tokens present in a place at the same time. Indeed, in the UML2AD specification, pins can hold multiple tokens, and can then have properties like an upper bound on the number of tokens or a consuming order (a FIFO, a pipe). In our subset any activity node may only hold one token at a time. The concept of *streaming*, an optional property of pins, is also left out,
- *token-based decisions* is a restriction on guards conditions. Whereas the general specification allows them to use data from the activity context, we restrict conditions to data carried by the incoming tokens.

AD-S: visual notation and concrete syntax

We use the visual notation and XML-based concrete syntax of UML2AD. Figure 4.6 shows the visual notation of most allowed constructs in AD-S.

4.2.2 AD-S: Constrained Object Model

We define AD-S through a constrained object model using the Z relational language. In order to facilitate reading, we also present classes, relations and attributes through UML class diagrams and provide descriptive texts. The AD-S elements that do not exist in UML2AD are marked with as "extension" in the UML class diagrams.

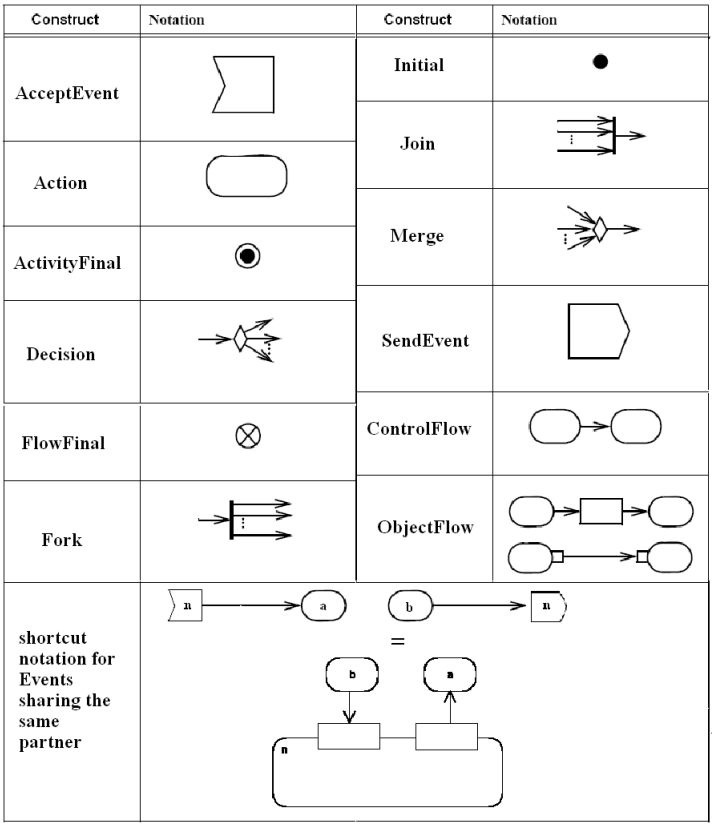


Figure 4.6: AD-S visual notation

Classes and Attributes

We define in Z a top class regrouping AD-S elements:

$$WFObject == UNIVERSE$$
$$WFClass == \mathbb{P}\ WFObject$$

Activity groups ActivityGroups are containers for ActivityNodes and ActivityEdges. GeneralGroups do not have other semantics, but InterruptibleActivityRegions are special groups used to model the external choice workflow pattern. As explained earlier, each node and edge has a boolean attribute called Active defining whether the element is part of the expected execution path(s)[2]

[2]In this document we often use, for clarity reasons, the same name for different attributes or relations. Since this is not allowed in Z, it does not reflect the actual definitions and is achieved using document-purpose renaming.

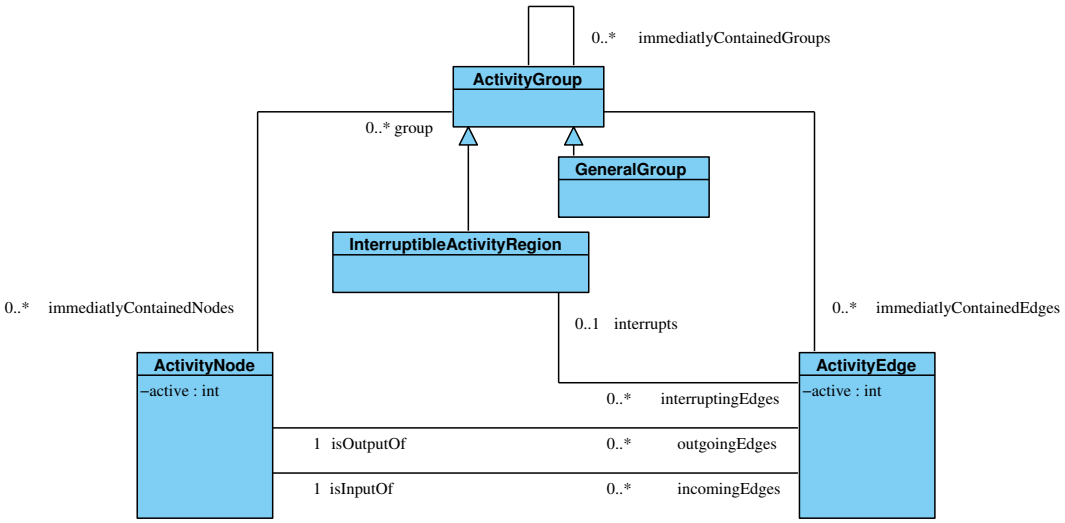


Figure 4.7: AD-S COM: Activity Groups

$$
\begin{array}{|l}
ActivityGroup : WFClass \\
GeneralGroup : WFClass \\
InterruptibleActivityRegion : WFClass \\
ActivityNode : WFClass \\
ActivityEdge : WFClass \\
\hline
\langle GeneralGroup, InterruptibleActivityRegion \rangle \ \mathsf{partition} \ ActivityGroup
\end{array}
$$

$$
\begin{array}{|l}
Boolean : \mathbb{N}
\end{array}
$$

$$
\begin{array}{|l}
active : ActivityNode \longrightarrow \mathbb{N} \\
active : ActivityEdge \longrightarrow \mathbb{N}
\end{array}
$$

Activity edges ActivityEdges are either ObjectFlows or ControlFlows, respectively allowing the data tokens flow and the control tokens flow between activity nodes.

$$
\begin{array}{|l}
ObjectFlow : WFClass \\
ControlFlow : WFClass \\
\hline
\langle ObjectFlow, ControlFlow \rangle \ \mathsf{partition} \ ActivityEdge
\end{array}
$$

$$
\begin{array}{|l}
Guard : ActivityEdge \longrightarrow String
\end{array}
$$

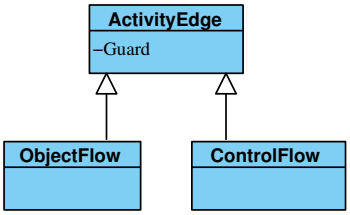


Figure 4.8: AD-S COM: Activity Edges

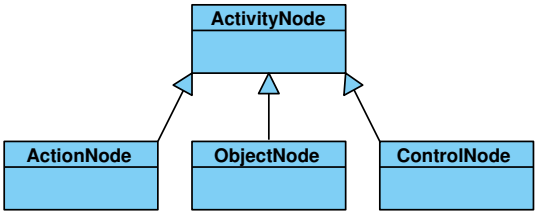


Figure 4.9: AD-S COM: Activity Nodes

Activity nodes ActivityNodes are divided into ActionNodes, ObjectNodes and ControlNodes.

$$\begin{array}{l} \mathit{ActionNode} : \mathit{WFClass} \\ \mathit{ObjectNode} : \mathit{WFClass} \\ \mathit{ControlNode} : \mathit{WFClass} \\ \hline \langle \mathit{ActionNode}, \mathit{ObjectNode}, \mathit{ControlNode} \rangle \ \mathsf{partition}\ \mathit{ActivityNode} \end{array}$$

Action nodes An ActionNode refers to an internal process being executed. An Event refers to a communication with an external agent, either the user or another SWS Event. A SendEvent is the action of sending a message, whereas an AcceptEvent is the action of waiting for a message. We also extended the UML2AD specifications with several stereotypes of ActionNodes that are useful in the context of SWSs. Operation is a predefined calculus on primitive types such as Integers. Transformation works on concepts from ontologies: Adaptation allows extraction/aggregation of concepts from/into composite concepts, whereas Mediation denotes a mediator between different concepts.

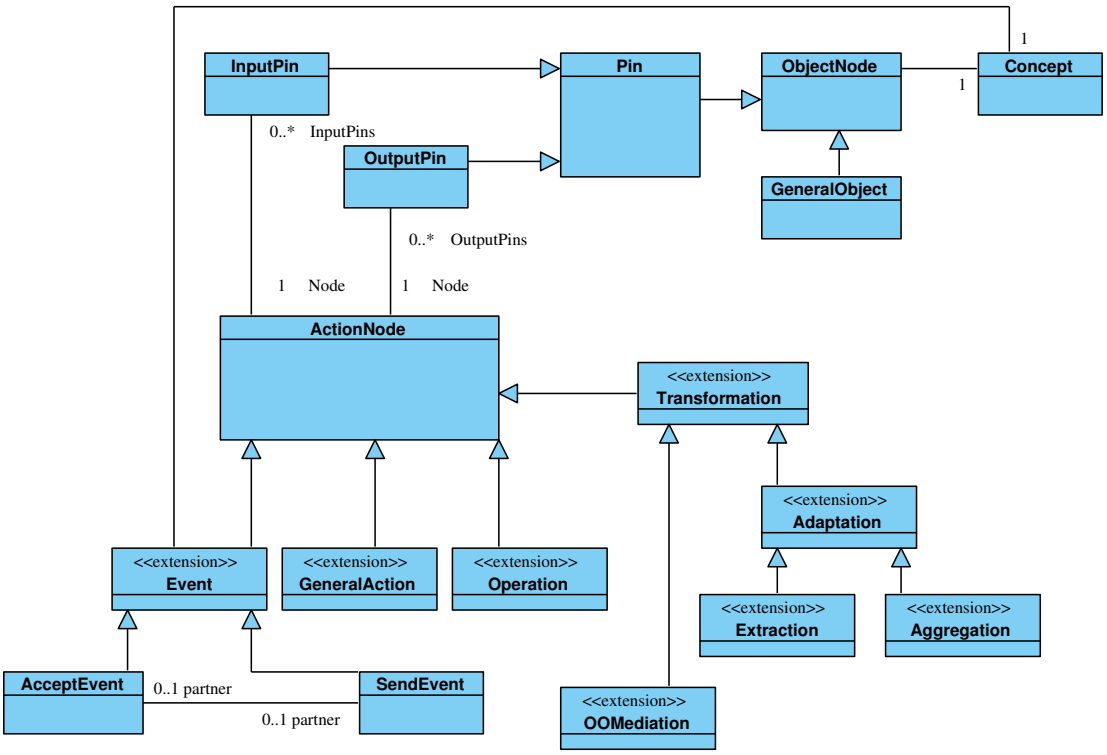


Figure 4.10: AD-S COM: Action and Object Nodes

$$
\begin{array}{l}
\mathit{GeneralAction} : \mathit{WFClass} \\
\mathit{Event} : \mathit{WFClass} \\
\mathit{AcceptEvent} : \mathit{WFClass} \\
\mathit{SendEvent} : \mathit{WFClass} \\
\mathit{Operation} : \mathit{WFClass} \\
\mathit{Transformation} : \mathit{WFClass} \\
\mathit{Mediation} : \mathit{WFClass} \\
\mathit{Adaptation} : \mathit{WFClass} \\
\mathit{Extraction} : \mathit{WFClass} \\
\mathit{Aggregation} : \mathit{WFClass} \\
\hline
\langle \mathit{GeneralAction}, \mathit{Event}, \mathit{Operation}, \mathit{Transformation} \rangle \ \mathsf{partition}\ \mathit{ActionNode} \\
\langle \mathit{AcceptEvent}, \mathit{SendEvent} \rangle \ \mathsf{partition}\ \mathit{Event} \\
\langle \mathit{Mediation}, \mathit{Adaptation} \rangle \ \mathsf{partition}\ \mathit{Transformation} \\
\langle \mathit{Extraction}, \mathit{Aggregation} \rangle \ \mathsf{partition}\ \mathit{Adaptation}
\end{array}
$$

$$
\mathit{URI} : \mathit{Mediation} \longrightarrow \mathit{String}
$$

Object nodes ObjectNodes define the concepts that incoming or outgoing data tokens can carry. Pins are special ObjectNodes attached to ActionNodes.

$$
\begin{array}{|l}
\textit{GeneralObject} : \textit{WFClass} \\
\textit{Pin} : \textit{WFClass} \\
\textit{InputPin} : \textit{WFClass} \\
\textit{OutputPin} : \textit{WFClass} \\
\hline
\langle \textit{GeneralObject}, \textit{Pin} \rangle \text{ partition } \textit{ObjectNode} \\
\langle \textit{InputPin}, \textit{OutputPin} \rangle \text{ partition } \textit{Pin}
\end{array}
$$

Concepts Concepts are taken from an imported ontology, they define the type of a given data flow.

$$
\begin{array}{|l}
\textit{Concept} : \textit{WFClass}
\end{array}
$$

$$
\begin{array}{|l}
\textit{name} : \textit{Concept} \longrightarrow \textit{String}
\end{array}
$$

Control nodes ControlNodes allow for workflow logics. A Fork duplicates an incoming token to outgoing edges, allowing for parralelization of activities. A Join is a strict synchronization fired when tokens are present in all its incoming edges. A Merge is a synchronization fired by any incoming token. A Decision produces a token in one (and only one) of its outgoing edges depending on the value of its incoming token. An Initial produces a token at the start of the activity. A FlowFinal consumes incoming tokens, whereas an ActivityFinal stops the entire activity (deletes all tokens) whenever it consumes a token.

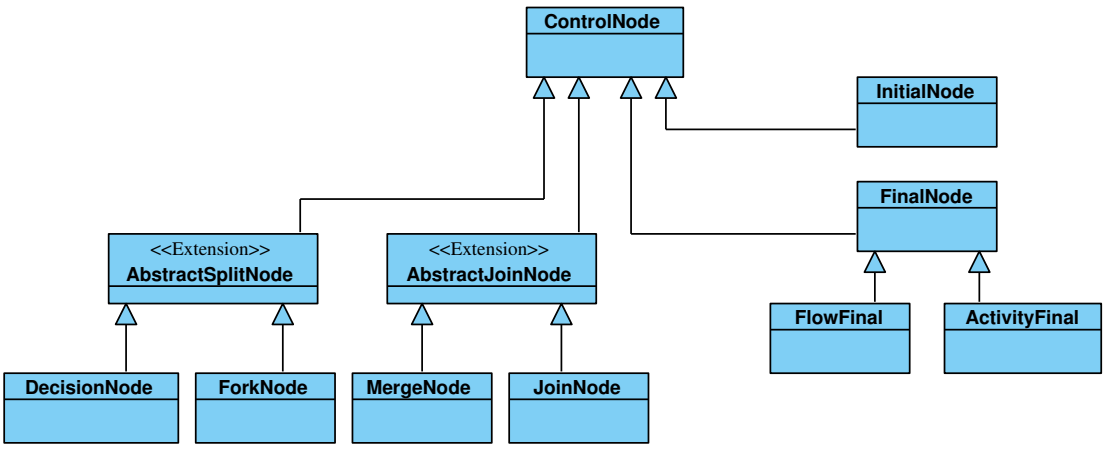


Figure 4.11: AD-S COM: Control Nodes

$$
\begin{array}{|l}
AbstractSplit : WFClass \\
AbstractJoin : WFClass \\
Join : WFClass \\
Merge : WFClass \\
Fork : WFClass \\
Decision : WFClass \\
Initial : WFClass \\
Final : WFClass \\
FlowFinal : WFClass \\
ActivityFinal : WFClass \\
\hline
\langle AbstractSplit, AbstractJoin, Initial, Final \rangle \text{ partition } ControlNode \\
\langle Join, Merge \rangle \text{ partition } AbstractJoin \\
\langle Fork, Decision \rangle \text{ partition } AbstractSplit \\
\langle FlowFinal, ActivityFinal \rangle \text{ partition } Final
\end{array}
$$

Relations

$$
\begin{array}{|l}
immediatlyContainedGroups : ActivityGroup \longrightarrow \mathbb{P}\, ActivityGroup
\end{array}
$$

$$
\begin{array}{|l}
immediatlyContainedEdges : ActivityGroup \longrightarrow \mathbb{P}\, ActivityEdge \\
group : ActivityEdge \longrightarrow ActivityGroup \\
\hline
\forall\, ag : ActivityGroup \bullet \\
ag.immediatlyContainedEdges = \{e : ActivityEdge \mid e.group = ag\}
\end{array}
$$

$$
\begin{array}{|l}
interruptingEdges : InterruptibleActivityRegion \longrightarrow \mathbb{P}\, ActivityEdge \\
interrupts : ActivityEdge \nrightarrow ActivityGroup \\
\hline
\forall\, ig : InterruptibleActivityRegion \bullet \\
ig.interruptingEdges = \{e : ActivityEdge \mid e.interrupts = ig\}
\end{array}
$$

$$
\begin{array}{|l}
immediatlyContainedNodes : ActivityGroup \longrightarrow \mathbb{P}\, ActivityNode \\
group : ActivityNode \longrightarrow ActivityGroup \\
\hline
\forall\, ag : ActivityGroup \bullet \\
ag.immediatlyContainedNodes = \{n : ActivityNode \mid n.group = ag\}
\end{array}
$$

$$
\begin{array}{|l}
incomingEdges : ActivityNode \longrightarrow \mathbb{P}\, ActivityEdge \\
isInputOf : ActivityEdge \longrightarrow ActivityNode \\
outgoingEdges : ActivityNode \longrightarrow \mathbb{P}\, ActivityEdge \\
isOutputOf : ActivityEdge \longrightarrow ActivityNode \\
\hline
\forall\, n : ActivityNode \bullet n.incomingEdges = \{e : ActivityEdge \mid e.isInputOf = n\} \\
\forall\, n : ActivityNode \bullet n.outgoingEdges = \{e : ActivityEdge \mid e.isOutputOf = n\}
\end{array}
$$

$$
\begin{array}{|l}
node : Pin \longrightarrow ActionNode \\
inputPins : ActionNode \longrightarrow \mathbb{P}\ InputPin \\
outputPins : ActionNode \longrightarrow \mathbb{P}\ OutputPin \\
\hline
\forall\, n : ActionNode \bullet n.inputPins = \{ip : InputPin \mid ip.node = n\} \\
\forall\, n : ActionNode \bullet n.outputPins = \{op : OutputPin \mid op.node = n\}
\end{array}
$$

$$
\begin{array}{|l}
concept : ObjectNode \longrightarrow Concept
\end{array}
$$

In our subset, an Event has a concept just as an ObjectNode does. It defines which type of message is sent or received.

$$
\begin{array}{|l}
concept : Event \longrightarrow Concept
\end{array}
$$

$$
\begin{array}{|l}
concept : ObjectFlow \longrightarrow Concept
\end{array}
$$

Events from different agents are not connected through edges. Instead, the partner relation of an orchestration Event defines which Event is supposed to receive or send the expected message in the participant SWS choreography.

$$
\begin{array}{|l}
partner : AcceptEvent \rightarrowtail\!\!\!\!\!\!\rightarrow SendEvent \\
partner : SendEvent \rightarrowtail\!\!\!\!\!\!\rightarrow AcceptEvent \\
\hline
\forall\, ae : AcceptEvent\ ;\ se : SendEvent \mid ae.partner = se \bullet se.partner = ae
\end{array}
$$

Predefined Objects

The User group contains the client choreography, in other words the user available inputs and required output messages. The Orchestration group will contain all the internal elements dynamically added to the composite workflow. Each participant SWS choreography has its own ActivityGroup.

$$
\begin{array}{|l}
User : ActivityGroup \\
Orchestration : ActivityGroup
\end{array}
$$

Constraints

Constraints define the requirements for a valid composite workflow. There are several types of constraints, from syntactical ones giving bounds to cardinalities to executional ones controlling the propagation of the Active attribute.

Cardinality constraints

- Control nodes

$$\forall\, x : AbstractSplit \bullet \#(x.incomingEdges) = 1$$
$$\forall\, x : AbstractJoin \bullet \#(x.outgoingEdges) = 1$$
$$\forall\, x : Initial \bullet x.incomingEdges = \varnothing$$
$$\forall\, x : Final \bullet x.outgoingEdges = \varnothing$$

- All ActivityNodes have at least one incoming edge (or one input pin for ActionNodes) with the following exceptions:
  InitialNodes, from their execution semantics,
  AcceptEvents, as in our context they are triggered by a corresponding SendEvent,
  SendEvents in the User group, as they represent the user input messages.
  As a consequence, note that activity propagation can only stop at those ActivityNodes.

$$\begin{aligned}
&\forall\, c : ControlNode \mid c.active = 1 \wedge c \notin Initial \bullet \\
&\quad \#(c.incomingEdges) \geq 1 \\
&\forall\, o : ObjectNode \mid o.active = 1 \bullet \\
&\quad \#(o.incomingEdges) \geq 1 \\
&\forall\, a : ActionNode \mid a.active = 1 \\
&\quad \wedge\, a \notin AcceptEvent \wedge a \notin SendEvent \bullet \\
&\qquad \#(a.incomingEdges) \geq 1 \vee \#(a.inputPins) \geq 1 \\
&\forall\, se : SendEvent \mid se.active = 1 \wedge se.group \neq User \bullet \\
&\quad \#(a.incomingEdges) \geq 1 \vee \#(a.inputPins) \geq 1 \\
&\forall\, ae : AcceptEvent \bullet ae.inputPins = \varnothing \\
&\forall\, se : SendEvent \bullet se.outputPins = \varnothing
\end{aligned}$$

- Object nodes

$$\forall\, ip : InputPin \bullet ip.outgoingEdges = \varnothing \wedge \#(ip.incomingEdges) = 1$$
$$\forall\, op : OutputPin \bullet op.incomingEdges = \varnothing \wedge \#(op.outgoingEdges) = 1$$
$$\forall\, go : GeneralObject \bullet \#(go.incomingEdges) = 1 \wedge \#(go.outgoingEdges) = 1$$

Flow constraints The following constraints specify which type of ActivityEdges can be connected to different types of nodes.

- Control flows

$$\begin{aligned}
&\forall\, cf : ControlFlow \bullet \\
&\quad cf.isInputOf \notin ObjectNode \wedge cf.isOutputOf \notin ObjectNode
\end{aligned}$$

- Object flows

$$\begin{aligned}
&\forall f : ObjectFlow \bullet \\
&\quad f.isInputOf \in ObjectNode \\
&\quad \vee f.isInputOf \in Decision \\
&\quad \vee f.isInputOf \in Merge \\
&\quad \vee f.isInputOf \in Fork \\
&\quad \vee f.isInputOf \in Join \\
&\forall f : ObjectFlow \bullet \\
&\quad f.isOutputOf \in ObjectNode \\
&\quad \vee f.isOutputOf \in Decision \\
&\quad \vee f.isOutputOf \in Merge \\
&\quad \vee f.isOutputOf \in Fork \\
&\quad \vee f.isOutputOf \in Join
\end{aligned}$$

- Control nodes

$$\begin{aligned}
&\forall x : ActivityNode \mid x \in Decision \cup Merge \bullet \\
&\quad (x.incomingEdges \cup x.outgoingEdges) \subset ObjectFlow \vee \\
&\quad (x.incomingEdges \cup x.outgoingEdges) \subset ControlFlow
\end{aligned}$$

Concepts constraints The following constraints ensure the compatibility of the data (concepts) throughout the workflow. For now we only allow for strict equivalence.

- Object nodes

$$\begin{aligned}
&\forall o : ObjectNode \mid \#(o.incomingEdges) = 1 \bullet \\
&\{o.concept\} = o.incomingEdges \rightarrow concept \\
&\forall o : ObjectNode \mid \#(o.outgoingEdges) = 1 \bullet \\
&\{o.concept\} = o.outgoingEdges \rightarrow concept
\end{aligned}$$

- Events

$$\begin{aligned}
&\forall ae : AcceptEvent \bullet \\
&ae.concept = ae.partner.concept \\
&\forall ae : AcceptEvent \mid \#(ae.outputPins) = 1 \bullet \\
&\{ae.concept\} = ae.outputPins \rightarrow concept \\
&\forall se : SendEvent \bullet \\
&se.concept = se.partner.concept \\
&\forall se : SendEvent \mid \#(se.inputPins) = 1 \bullet \\
&\{se.concept\} = se.inputPins \rightarrow concept
\end{aligned}$$

- Control nodes

$$\begin{aligned}
&\forall m : Merge \bullet \#(m.incomingEdges \rightarrow concept) = 1 \\
&\quad \wedge m.incomingEdges \rightarrow concept = m.outgoingEdges \rightarrow concept
\end{aligned}$$

Activity propagation constraints The following constraints control the propagation of the active attribute, each one is based on the execution semantics of the workflow element.

- Activity edges

$$\forall\, e : \mathit{ActivityEdge} \mid e.\mathit{active} = 1 \bullet e.\mathit{isOutputOf}.\mathit{active} = 1$$

- Object nodes

$$\forall\, o : \mathit{ObjectNode}\ ;\ e : \mathit{ActivityEdge} \mid \\ o.\mathit{active} = 1 \land e \in o.\mathit{incomingEdges} \bullet e.\mathit{active} = 1$$

- Output pins

$$\forall\, op : \mathit{OutputPin} \mid op.\mathit{active} = 1 \bullet op.\mathit{node}.\mathit{active} = 1$$

- Action nodes

$$\forall\, a : \mathit{ActionNode}\ ;\ e : \mathit{ActivityEdge} \mid \\ a.\mathit{active} = 1 \land e \in a.\mathit{incomingEdges} \bullet e.\mathit{active} = 1 \\ \forall\, a : \mathit{ActionNode}\ ;\ ip : \mathit{InputPin} \mid \\ a.\mathit{active} = 1 \land ip \in a.\mathit{inputPins} \bullet ip.\mathit{active} = 1$$

- Events

$$\forall\, ae : \mathit{AcceptEvent} \mid ae.\mathit{active} = 1 \bullet ae.\mathit{partner}.\mathit{active} = 1$$

- Control nodes[3]

$$\forall\, s : \mathit{AbstractSplit}\ ;\ e : \mathit{ActivityEdge} \mid s.\mathit{active} = 1 \land e \in s.\mathit{incomingEdges} \bullet \\ \quad e.\mathit{active} = 1 \\ \forall\, j : \mathit{Join}\ ;\ e : \mathit{ActivityEdge} \mid j.\mathit{active} = 1 \land e \in j.\mathit{incomingEdges} \bullet \\ \quad e.\mathit{active} = 1 \\ \forall\, f : \mathit{Final}\ ;\ e : \mathit{ActivityEdge} \mid f.\mathit{active} = 1 \land e \in f.\mathit{incomingEdges} \bullet \\ \quad e.\mathit{active} = 1 \\ \forall\, m : \mathit{Merge} \mid m.\mathit{active} = 1 \bullet \\ \mathit{count}\ (m.\mathit{incomingEdges} \rightsquigarrow \mathit{active})\ 1 \geq 1$$

[3]The operator $\mathit{count}\ (B)\ x$ is used to count elements with value x in a given bag B.

Miscellaneous constraints

- Interrupting edges have their source node in the interrupted group and their target node outside this group

$$\forall e : ActivityEdge\ ;\ ig : InterruptibleActivityRegion \mid ig = e.interrupts \bullet \\ e.isOutputOf.group = ig \wedge \\ e.isInputOf.group \neq ig$$

- All User and external SWSs Events have their partner in the Orchestration

$$\forall ae : AcceptEvent \bullet ae.group = Orchestration \\ \quad \vee\ ae.partner.group = Orchestration \\ \forall se : SendEvent \bullet se.group = Orchestration \\ \quad \vee\ se.partner.group = Orchestration$$

- All nodes and edges in the Orchestration are active

$$\forall n : ActivityNode \mid n.group = Orchestration \bullet n.active = 1 \\ \forall e : ActivityEdge \mid e.group = Orchestration \bullet e.active = 1$$

4.2.3 Computing a composite workflow

Based on the presented COM, we can use a configuration tool to compose workflows. We provide the following inputs to the configurator:

- the COM,
- a set of available choreographies as objects of the COM,
- the corresponding set of ontologies as objects of the COM,
- a set of available mediators as Mediation objects,
- a set of user available input messages as SendEvent objects of the user's group,
- a set of user required output messages as active AcceptEvent objects of the user's group. These are the root objects.

The configurator then builds a composite workflow containing all required choreographies and for which all created workflow constructs are added to the predefined Orchestration group.

Definition 4.2.1 *A composite workflow cw is composed of:*
An orchestration group o, a user group u, a set of choreography groups $C = c_0, \ldots, c_j$ *(all consumed SWSs).*

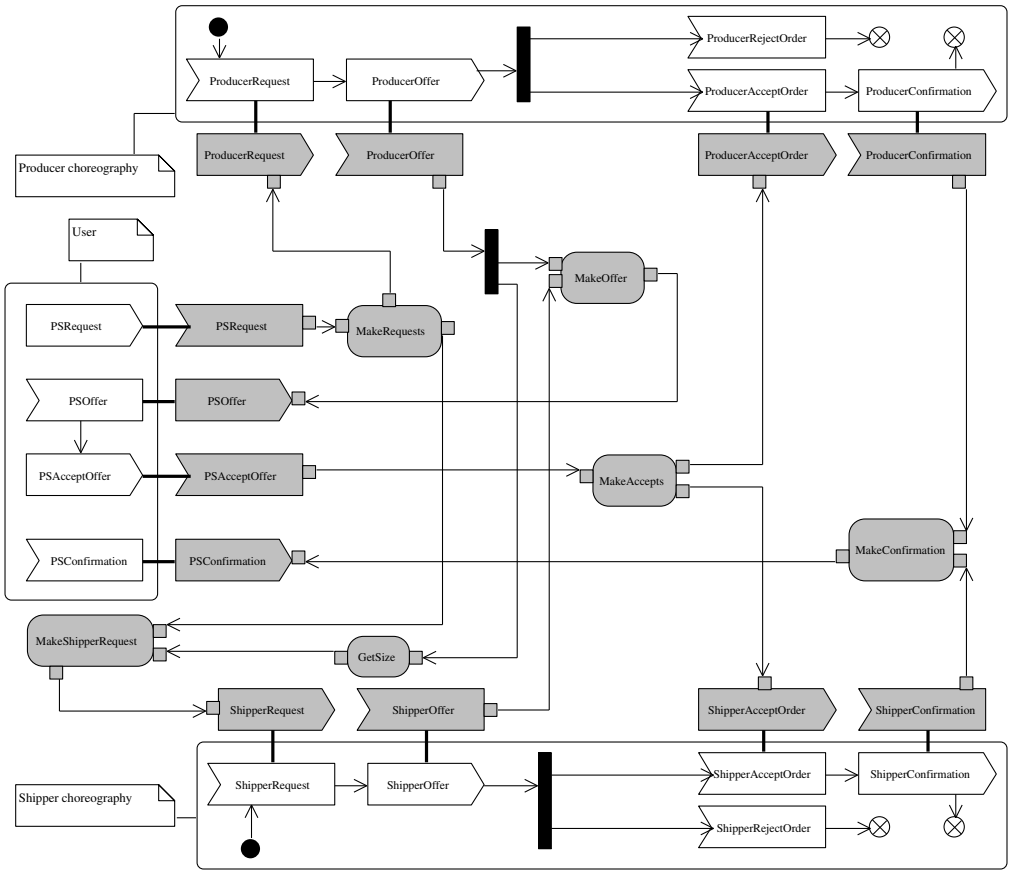


Figure 4.12: AD-S: composite workflow for the producer-shipper-a senario (bold edges represent partner relations)

Figure 4.12 is an example of a simple composite workflow configured using the AD-S model. In this example, the user requires an aggregated confirmation of an order from both a producer and a shipper. The user interaction pattern which allows to accept offers is made explicit in the User group. There is also a required interleaving between the shipper and the producer since the shipper request contains information from the producer offer. Action nodes operating a transformation are given as available mediators.
Another example has been presented at the beginning of the Chapter with the NMPC-bundle scenario. Figure 4.5 shows an orchestration extracted from a computed composite workflow (the extraction which removed consumed choreographies and added some workflow constructs is detailed in a later section).

Composite workflows execution properties The propagation of the Active attribute ensures that in the composite workflow there exists at least one valid execution path to the user's objectives. Indeed, for each node or edge having a *true* value for this attribute, the constraints guaranty that sufficient firing conditions are met. However, it is not guaranteed that the path will be taken at execution because of possible DecisionNodes in the consumed choreographies. Indeed, some guard conditions such as value of tokens at execution may not be reasoned about during design-time. For instance consider the choreography of Figure 4.13. The guard condition on the DecisionNode depends on the token value at execution (it can be a user's input). Therefore the required behaviour may not be achieved during execution. As a consequence, there is another property on the composite workflows. The solutions may contain objects which are not part of the expected behaviour, denoted by a *false* value for their Active attribute. Those unused workflow parts are however only in the consumed choreographies groups, whereas all objects in the orchestration are active.

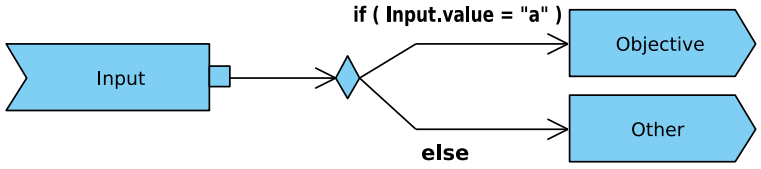


Figure 4.13: AD-S: Alternative paths with execution-time conditions

4.3 Configurable Composition Requests: Composition Goals

The introduction of a composition request language is motivated by several issues that arise in process-level composition.

Complex composition requirements As pointed in the previous Chapter, in the case of design-time composition, the request to a composer should allow to express additional requirements on the desired composite SWS (besides the available inputs and required outputs). We call such composition requests "composition goals". These requirements can be expressed at the capability (or goal) level in order to abstract from workflows behaviour and enable interaction with discovery tools.

Issues in process-only composition Experiments on our process-level composer also revealed issues that cannot be solved without goal-level specifications. On the one

hand, the library's size of available SWS choreographies quickly becomes a problem because of the combinatorial explosion. Interaction with discovery allows to significantly reduce the number of potential candidates.
On the other hand, the composer sometimes cannot make correct inferences based solely on messages types. For instance, consider a service offering plane reservation with departure and arrival city with the same message type as input and output, and a service offering accommodation with the same city message type as input. Having the accommodation in the departure or in the arrival city are both viable options for the composer. The request to the composer must then be able to remove those ambiguities by placing symbolic links between messages playing the same *role*.

4.3.1 A language for the specification of composition goals

We define a composition goal language (CG), at the abstract level of goals. A modular translation to the AD-S elements and constraints allows to take into account the requirements when generating the composite workflow. The language is able to express:

- constraints on the non-functional properties (NFPs) of composed web services,
- constraints on the concepts carried by messages,
- constraints on the data flow. In particular symbolic links between messages, but also defining alternative paths (for instance, if I travel by car I want a parking in my hotel),
- constraints on the control flow for temporal requirements (for instance receive an order before I send a payment).

The language is defined as a COM in Subsection 4.3.2, hence allowing for assisted, semi-automatic or automatic composition of the request (i.e goal-level composition). The language is also given a graphical representation in Subsection 4.3.2 through a concrete syntax based on UML2 activity diagrams (only the syntax, not the semantics of UML2AD).

4.3.2 CG: Constrained Object Model

Again, we describe our constrained object model using the Z relational language. We call it the CG-model in the following. In order to facilitate reading, we also present classes, relations and attributes through UML class diagrams and descriptive texts.

Classes and Attributes

We define in Z a top class regrouping CG elements:

$$CGObject == UNIVERSE$$
$$CGClass == \mathbb{P}\, CGObject$$

A CompositionGoal is the core element. The contained AtomicGoals represent the desired (or required) capabilities, which will be used by discovery to find appropriate SWSs. Available AtomicGoals are taken from a goal library.

$$AbstractGoal : CGClass$$
$$AtomicGoal : CGClass$$
$$CompositionGoal : CGClass$$
$$\langle AtomicGoal, CompositionGoal \rangle \text{ partition } AbstractGoal$$

$$name : AbstractGoal \longrightarrow String$$
$$name : AtomicGoal \longrightarrow String$$

Roles are the abstraction of messages at the goal-level. InternalRoles represent orchestration's internal messages.

$$Role : CGClass$$
$$InputRole : CGClass$$
$$OutputRole : CGClass$$
$$InternalRole : CGClass$$
$$\langle InputRole, OutputRole, InternalRole \rangle \text{ partition } Role$$

$$name : Role \longrightarrow String$$

Concepts, in the same way as messages, represent the type of a Role. However here a concept can be a CompositeConcept containing AtomicConcepts, and a concept can have subconcepts. This is a modeling of composites and inheritance that allows for simple data reasoning as will be explained later.

$$AbstractConcept : CGClass$$
$$AtomicConcept : CGClass$$
$$CompositeConcept : CGClass$$
$$\langle AtomicConcept, CompositeConcept \rangle \text{ partition } AbstractConcept$$

$$name : AbstractConcept \longrightarrow String$$

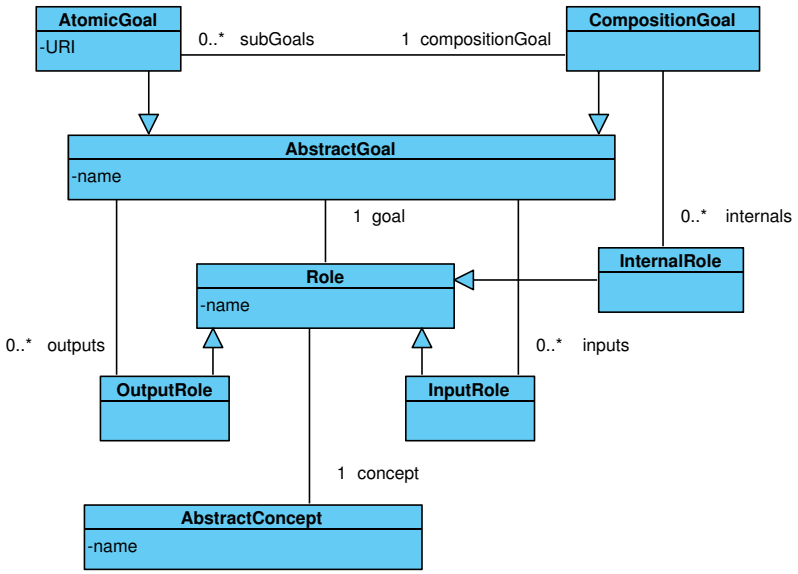


Figure 4.14: CG COM: Goals and Roles

PropertyConstraints apply to services NFPs, whereas ValueConstraints,DataflowConstraints and ControlflowConstraints apply to Roles.

$$
\begin{array}{|l}
\mathit{Constraint} : \mathit{CGClass} \\
\mathit{ValueConstraint} : \mathit{CGClass} \\
\mathit{PropertyConstraint} : \mathit{CGClass} \\
\mathit{ControlFlowConstraint} : \mathit{CGClass} \\
\mathit{DataflowConstraint} : \mathit{CGClass} \\
\hline
\langle \mathit{ValueConstraint}, \mathit{PropertyConstraint}, \mathit{ControlFlowConstraint}, \mathit{DataflowConstraint} \rangle \\
\quad \text{partition } \mathit{Constraint}
\end{array}
$$

$$
\begin{array}{|l}
\mathit{name} : \mathit{Constraint} \longrightarrow \mathit{String}
\end{array}
$$

IdentityFlow expresses the semantic equivalence of two roles, hence later grounded to the same message in the composite workflow. MergeFlow and DecisionFlow allow for alternative paths to be defined. OperationFlow and AdaptationFlow allow for transformations to be explicitly created beforehand.

$$
\begin{array}{|l}
IdentityFlow : CGClass \\
MergeFlow : CGClass \\
DecisionFlow : CGClass \\
ActionFlow : CGClass \\
OperationFlow : CGClass \\
AdaptationFlow : CGClass \\
\hline
\langle IdentityFlow, MergeFlow, DecisionFlow, ActionFlow \rangle \\
\quad \mathsf{partition}\ DataflowConstraint \\
\langle OperationFlow, AdaptationFlow \rangle \\
\quad \mathsf{partition}\ ActionFlow
\end{array}
$$

$$
\begin{array}{|l}
MediationFlow : CGClass \\
AggregationFlow : CGClass \\
ExtractionFlow : CGClass \\
\hline
\langle MediationFlow, AggregationFlow, ExtractionFlow \rangle\ \mathsf{partition}\ AdaptationFlow
\end{array}
$$

$$
\begin{array}{|l}
URI : MediationFlow \longrightarrow String
\end{array}
$$

A ValueConstraint expresses a requirement on a concept carried by a Role, either to one Role (UnaryValueConstraint) or to multiple roles (RelationalValueConstraint). For now the only supported constraints are refinements of a concept to one of its subconcepts, whereas operators for RelationalValueConstraints are restricted to equivalence between two concepts.

$$
\begin{array}{|l}
UnaryValueConstraint : CGClass \\
RelationalValueConstraint : CGClass \\
\hline
\langle UnaryValueConstraint, RelationalValueConstraint \rangle\ \mathsf{partition}\ ValueConstraint
\end{array}
$$

Guards express conditions of alternative paths. A DecisionTarget is assigned to each path, and points to both the target (a Role) and the condition (a Statement)

$$
\begin{array}{|l}
Statement : CGClass \\
Guard : CGClass \\
DecisionTarget : CGClass
\end{array}
$$

Relations

$$
\begin{array}{|l}
subGoals : CompositionGoal \longrightarrow \mathbb{P}\ AtomicGoal \\
compositionGoal : AtomicGoal \longrightarrow CompositionGoal \\
\hline
\forall\ cg : CompositionGoal \bullet cg.subGoals = \\
\quad \{ag : AtomicGoal \mid ag.compositionGoal = cg\}
\end{array}
$$

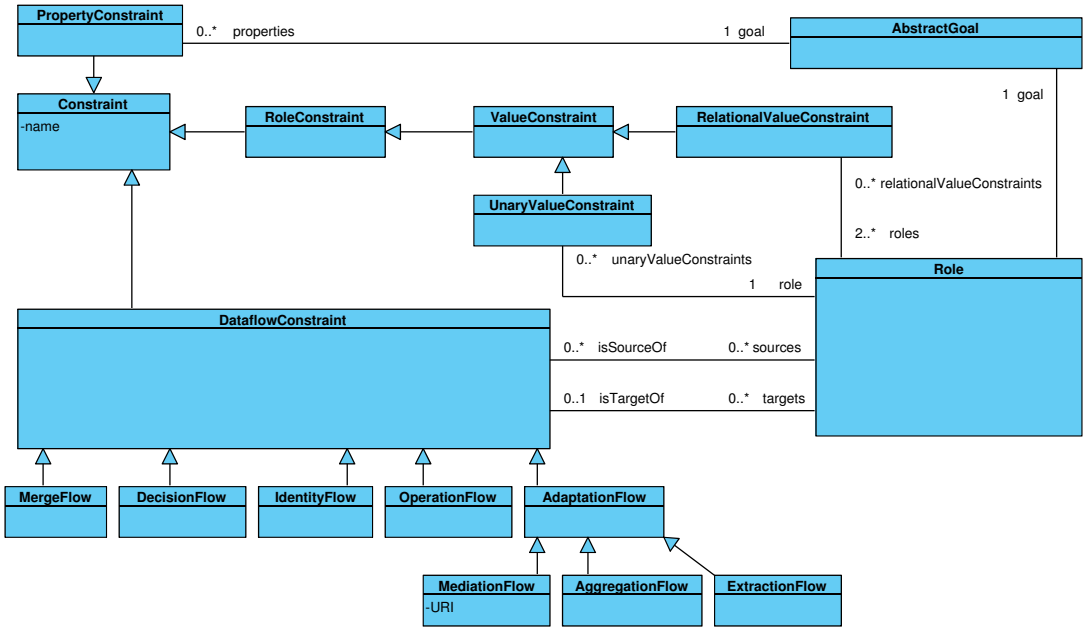


Figure 4.15: CG COM: Constraints

$$
\begin{array}{l}
goal : Role \longrightarrow AbstractGoal \\
inputs : AbstractGoal \longrightarrow \mathbb{P}\ InputRole \\
outputs : AbstractGoal \longrightarrow \mathbb{P}\ OutputRole \\
internals : CompositionGoal \longrightarrow \mathbb{P}\ InternalRole \\
\hline
\forall\, ag : AbstractGoal \bullet ag.inputs = \{r : InputRole \mid r.goal = ag\} \\
\forall\, ag : AbstractGoal \bullet ag.outputs = \{r : OutputRole \mid r.goal = ag\} \\
\forall\, cg : CompositionGoal \bullet cg.internals = \{r : InternalRole \mid r.goal = cg\}
\end{array}
$$

$$
\begin{array}{l}
concept : Role \longrightarrow AbstractConcept \\
subConcepts : AbstractConcept \longrightarrow \mathbb{P}\ AbstractConcept \\
subConcepts : AbstractConcept \leftrightarrow AbstractConcept \\
super : AbstractConcept \nrightarrow AbstractConcept \\
contains : CompositeConcept \longrightarrow \mathbb{P}\ AbstractConcept \\
contains : CompositeConcept \leftrightarrow AbstractConcept \\
\hline
\forall\, cc : CompositeConcept \bullet contains(cc) = contains(\!|\{cc\}|\!) \\
\forall\, c : AbstractConcept \bullet subConcepts(c) = subConcepts(\!|\{c\}|\!) \\
\forall\, ac : AbstractConcept \bullet ac.subConcepts = \\
\qquad \{ac2 : AbstractConcept \mid ac2.super = ac\} \\
\forall\, cc : CompositeConcept \bullet \#(cc.contains) \geq 1
\end{array}
$$

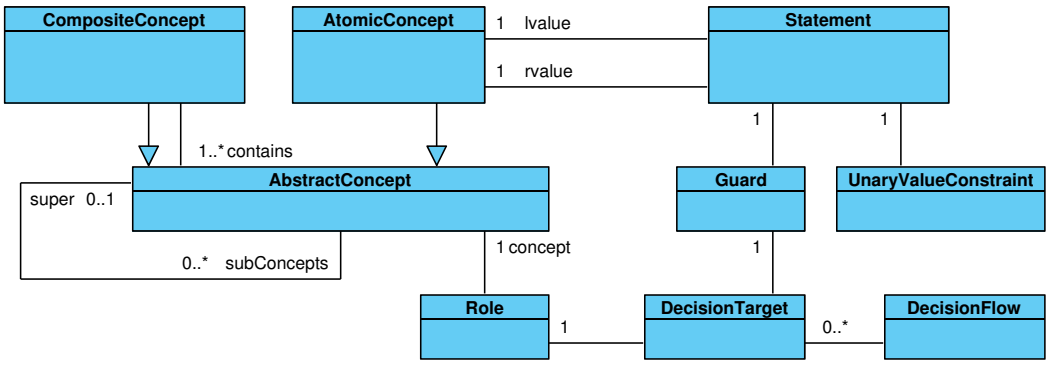


Figure 4.16: CG COM: Concepts and Statements

$$
\begin{array}{|l}
sources : DataflowConstraint \longrightarrow \mathbb{P}\ Role \\
isSourceOf : Role \longrightarrow \mathbb{P}\ DataflowConstraint \\
targets : DataflowConstraint \longrightarrow \mathbb{P}\ Role \\
isTargetOf : Role \nrightarrow DataflowConstraint \\
role : UnaryValueConstraint \nrightarrow Role \\
unaryValueConstraints : Role \longrightarrow \mathbb{P}\ UnaryValueConstraint \\
roles : RelationalValueConstraint \longrightarrow \mathbb{P}\ Role \\
relationalValueConstraints : Role \longrightarrow \mathbb{P}\ RelationalValueConstraint \\
\hline
\forall\, df : DataflowConstraint \bullet df.sources = \{r : Role \mid df \in r.isSourceOf\} \\
\forall\, df : DataflowConstraint \bullet df.targets = \{r : Role \mid r.isTargetOf = df\} \\
\forall\, uv : UnaryValueConstraint \bullet \{uv.role\} = \\
\quad \{r : Role \mid uv \in r.unaryValueConstraints\} \\
\forall\, rv : RelationalValueConstraint \bullet rv.roles = \\
\quad \{r : Role \mid rv \in r.relationalValueConstraints\} \\
\forall\, rv : RelationalValueConstraint \bullet \#(rv.roles) \geq 2
\end{array}
$$

$$
\begin{array}{|l}
properties : AbstractGoal \longrightarrow \mathbb{P}\ PropertyConstraint
\end{array}
$$

$$
\begin{array}{|l}
lvalue : Statement \longrightarrow AbstractConcept \\
rvalue : Statement \longrightarrow AbstractConcept
\end{array}
$$

$$
\begin{array}{|l}
statement : UnaryValueConstraint \longrightarrow Statement \\
statement : Guard \longrightarrow Statement
\end{array}
$$

$$
\begin{array}{|l}
decisionTargets : DecisionFlow \longrightarrow \mathbb{P}\ DecisionTarget \\
guard : DecisionTarget \longrightarrow Guard \\
role : DecisionTarget \longrightarrow Role
\end{array}
$$

Constraints

Cardinality and relation constraints

$$\forall\, r : InputRole \bullet \#\{r.isTargetOf\} > 0$$
$$\forall\, r : InternalRole \bullet \#\{r.isTargetOf\} > 0$$
$$\forall\, n : DecisionFlow \bullet \#\{n.sources\} = 1$$
$$\forall\, n : MergeFlow \bullet \#\{n.targets\} = 1$$

Role constraints Sources of FlowConstraints are either OutputRoles or InternalRoles, targets of FlowConstraints are either InputRoles or InternalRoles:

$$\forall\, r : Role \mid \#(r.isSourceOf) > 0 \bullet r \in OutputRole \lor r \in InternalRole$$
$$\forall\, r : Role \mid \#(\{r.isTargetOf\}) > 0 \bullet r \in InputRole \lor r \in InternalRole$$

Concepts compatibility constraints

- Sources and targets of IdentityFlow, DecisionFlow and MergeFlow must share the same concept:

$$\forall\, n : DataflowConstraint \mid$$
$$(n \in IdentityFlow \lor n \in MergeFlow \lor n \in DecisionFlow) \bullet$$
$$\quad sources(n) \rightarrow concept = targets(n) \rightarrow concept$$

- ExtractionFlow has a single source which concept is composite, and targets are concepts included in the source:

$$\forall\, n : ExtractionFlow \bullet$$
$$\#\{n.sources\} = 1$$
$$\land\ sources(n) \rightarrow concept \in \{CompositeConcept\}$$
$$\land\ targets(n) \rightarrow concept$$
$$\quad \subset sources(n) \rightarrow concept \rightarrow contains^{+}$$

- AggregationFlow has a single target which concept is composite, and sources are concepts included in the target:

$$\forall\, n : AggregationFlow \bullet$$
$$\#(n.targets) = 1$$
$$\land\ targets(n) \rightarrow concept \in \{CompositeConcept\}$$
$$\land\ sources(n) \rightarrow concept$$
$$\quad \subset targets(n) \rightarrow concept \rightarrow contains^{+}$$

Statements constraints

- Validity of Statements (restricted to concept refinement):

$$\forall\, a : Statement \bullet a.rvalue \in (a.lvalue.subConcepts^{+})$$

- Validity and propagation of UnaryValueConstraints:

$$\forall\, u : UnaryValueConstraint \\ \mid \{u.role.concept\} \cap AtomicConcept \neq \varnothing \bullet \\ u.statement.lvalue = u.role.concept$$

$$\forall\, u : UnaryValueConstraint\ ;\ cc : CompositeConcept \\ \mid u.role.concept = cc \bullet \\ u.statement.lvalue \in cc.contains^{*}$$

$$\forall\, dt : DecisionTarget \bullet \\ dt.guard.statement \in dt.role.unaryValueConstraints \rightarrow statement$$

$$\forall\, uvc1, uvc2 : UnaryValueConstraint \\ \mid uvc1.role = uvc2.role \\ \wedge\ uvc1.statement.lvalue = uvc2.statement.lvalue \bullet \\ uvc1.statement.rvalue = uvc2.statement.rvalue$$

Concrete syntax

The language is given a graphical representation through a concrete syntax based on stereotypes of the UML2 activity diagrams. UML2AD is chosen as a visual facility hence only the syntax and not the semantics of UML2AD are used. This visual notation allows for easier design, representation and XML import/export facilities to/from the composer. Figure 4.17 is a summary of this graphical representation.

4.3.3 Computing a composition goal

Based on the presented COM, we can use a configuration tool to generate composition goals. We provide the following inputs to the configurator:

- the COM,
- a set of available atomic goals and mediators (from a library) as objects of the COM,
- a set of ontologies as objects of the COM,
- a composition request as a partial composition goal (at least the composition goal's available InputRoles and required OutputRoles).

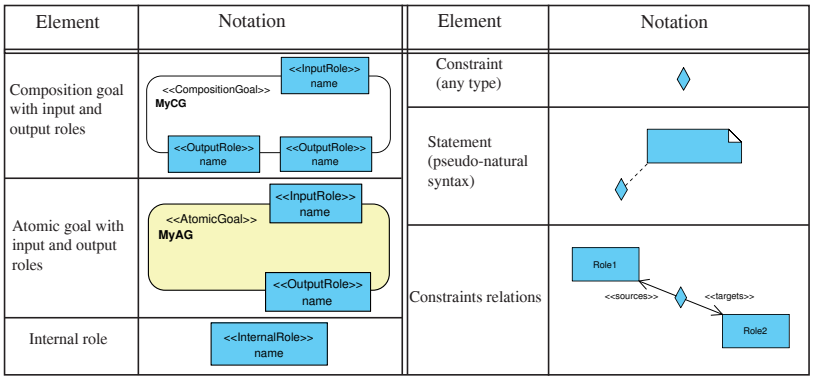


Figure 4.17: CG: concrete syntax based on UML2AD

The configurator builds a (completed) composition goal containing all required atomic goals as well as any necessary internal logic (ValueConstraints, FlowConstraints).
The example presented at the beginning of the Chapter in Figures 4.3 and 4.4 shows how a complex composition goal can be computed out of a composition request.
As there are no choreographies (hence no unuseful alternative paths), all elements in the configurated structure actively participate to the solution.

4.3.4 From composition goals to workflow configuration

The composition goals express requirements on the composite workflow. We present an extension of the AD-S model which allows to take into account the composition goal as additional model elements. A modular translation from objects of the CG-model to objects of the new AD-S model is then proposed, together with constraints implementing the semantics of the requirements.

Extension of the UML2AD-S model

Classes and Attributes WFCompositionGoal is the motherclass for the new elements.

$$WFCompositionGoal : WFClass$$

WFFlow is divided into IdentityFlow and ControlFlow only. The translation details how those two classes are sufficient for all types of FlowConstraints that can be expressed in the CG-model.

$$WFFlow : CGClass$$
$$WFIdentityFlow : CGClass$$
$$WFControlFlow : CGClass$$
$$\langle WFIdentityFlow, WFControlFlow \rangle \text{ partition } WFFlow$$

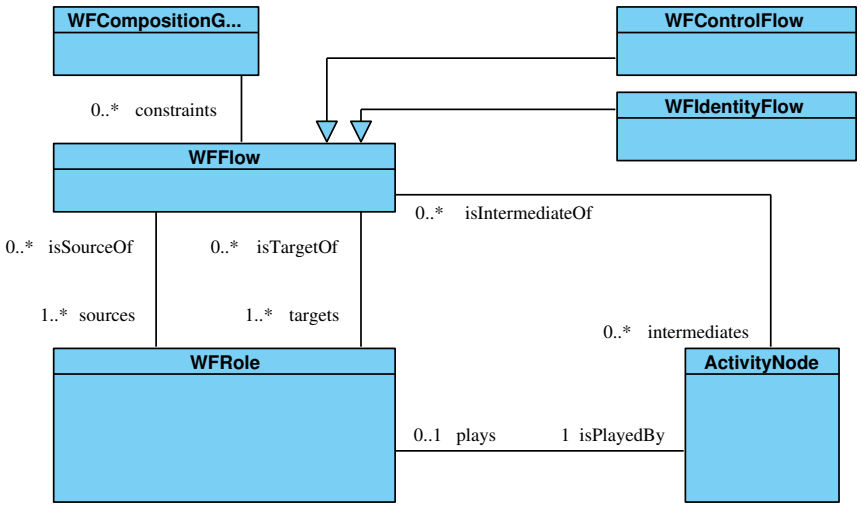


Figure 4.18: AD-S COM: CG extension

WFRoles are abstractions of messages just as in the CG-model.

$$WFRole : WFClass$$

Relations The (unique) WFCompositionGoal object contains a set of WFFlow.

$$constraints : WFCompositionGoal \longrightarrow \mathbb{P}\ WFFlow$$

A WFFlow applies on WFRoles through the *sources* and *targets* relations.

$$\begin{array}{l} targets : WFFlow \longrightarrow \mathbb{P}\ WFRole \\ sources : WFFlow \longrightarrow \mathbb{P}\ WFRole \\ isTargetOf : WFRole \longrightarrow \mathbb{P}\ WFFlow \\ isSourceOf : WFRole \longrightarrow \mathbb{P}\ WFFlow \\ \hline \forall\, fc : WFFlow \bullet fc.targets = \{r : WFRole \mid fc \in r.isTargetOf\} \\ \forall\, fc : WFFlow \bullet fc.sources = \{r : WFRole \mid fc \in r.isSourceOf\} \\ \forall\, fc : WFFlow \bullet \#(fc.targets) \geq 1 \\ \forall\, fc : WFFlow \bullet \#(fc.sources) \geq 1 \end{array}$$

A WFRole is played in the orchestration by one chosen ActivityNode.

$$\begin{array}{l} isPlayedBy : WFRole \nrightarrow ActivityNode \\ plays : ActivityNode \longrightarrow WFRole \\ \hline \forall\, r : WFRole\ ;\ n : ActivityNode \mid r.isPlayedBy = n \bullet n.plays = r \end{array}$$

An *intermediates* relation allows to relax constraints on the semantics of the IdentityFlow. This is further detailed in the translation.

$$
\begin{array}{|l}
intermediates : WFFlow \longrightarrow \mathbb{P}\, ActivityNode \\
isIntermediateOf : ActivityNode \longrightarrow \mathbb{P}\, WFFlow \\
\hline
\forall\, fc : WFFlow \bullet fc.intermediates = \{n : ActivityNode \mid fc \in n.isIntermediateOf\}
\end{array}
$$

Translation of CG objects to AD-S objects

We present the translation as Z constraints between the two models. However the implementation can be (and is) done with a straightforward linear algorithm: each object of the composition goal creates a set of related objects in the AD-S model.

Roles Each Role in the CG model creates a WFRole in the AD-S model.

$$
\begin{array}{|l}
tr : Role \longrightarrow WFRole
\end{array}
$$

$$\forall\, r : Role \bullet \exists\, r' : WFRole \bullet r.tr = r'$$

Identity Flows Each IdentityFlow in the CG model creates a WFIdentityFlow in the AD-S model. The sources and targets are the corresponding WFRoles.

$$
\begin{array}{l}
\forall\, if : IdentityFlow \bullet \exists\, if' : WFIdentityFlow \bullet \\
\quad if.sources \rightarrow tr = if'.sources \\
\quad \land\ if.targets \rightarrow tr = if'.targets
\end{array}
$$

Action Flows We transform an ActionFlow into a concrete ActionNode with equivalent semantics. WFIdentityFlows, WFInputRoles and WFOutputRoles are then created to link the new ActionNode to the ActionFlow InputRoles and OutputRoles. An Example of the operation translation is given in Figure 4.19.

$$
\begin{array}{|l}
tr : ActionFlow \longrightarrow ActionNode
\end{array}
$$

$$
\begin{array}{l}
\forall\, af : ActionFlow \bullet \exists\, af' : ActionNode \bullet af.tr = af' \\
\forall\, af : ActionFlow\ ;\ r : Role \mid r \in af.sources \bullet \\
\quad \exists\, ip : InputPin\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet \\
\qquad ip.node = af.tr \land r''.isPlayedBy = ip \\
\qquad \land\ r.tr \in if.sources \land \#(if.sources) = 1 \\
\qquad \land\ r'' \in if.targets \land \#(if.targets) = 1 \\
\forall\, af : ActionFlow\ ;\ r : Role \mid r \in af.targets \bullet \\
\quad \exists\, op : OutputPin\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet \\
\qquad op.node = af.tr \land r''.isPlayedBy = op \\
\qquad \land\ r.tr \in if.targets \land \#(if.targets) = 1 \\
\qquad \land\ r'' \in if.sources \land \#(if.sources) = 1
\end{array}
$$

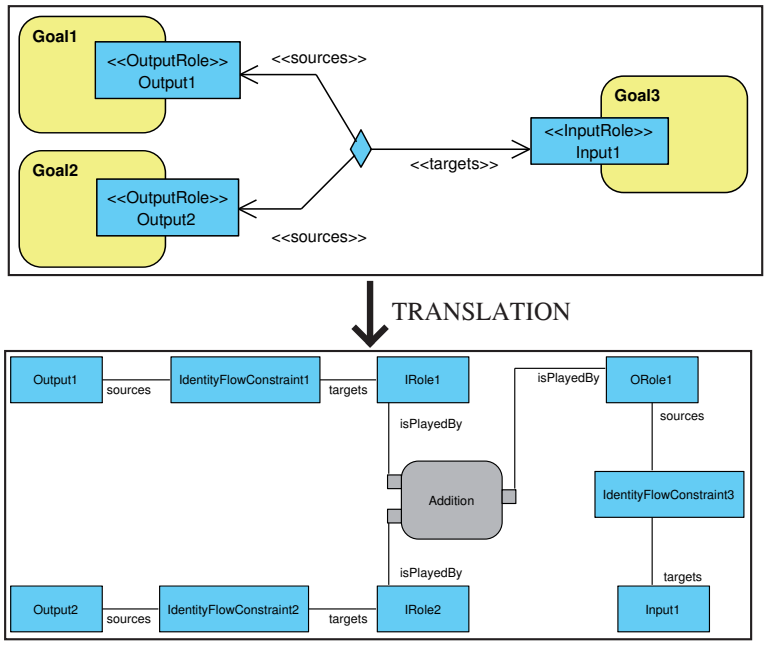


Figure 4.19: CG to AD-S: example of the Operation translation

Merge Flows Similarly to ActionFlows, we transform a MergeFlow into a concrete Merge node. We use auxiliary ObjectNodes to allow the linking of Roles. An Example of the merge translation is given in Figure 4.20.

$$| \quad tr : MergeFlow \longrightarrow Merge$$

$$\forall\, mf : MergeFlow \bullet \exists\, mf' : Merge \bullet mf.tr = mf'$$
$$\forall\, mf : MergeFlow\ ;\ r : Role \mid r \in mf.sources \bullet$$
$$\exists\, go : GeneralObject\ ;\ of : ObjectFlow\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet$$
$$of \in go.outgoingEdges \land of.isInputOf = mf.tr \land r''.isPlayedBy = go$$
$$\land\ r.tr \in if.sources \land \#(if.sources) = 1$$
$$\land\ r'' \in if.targets \land \#(if.targets) = 1$$
$$\forall\, mf : MergeFlow\ ;\ r : Role \mid r \in mf.sources \bullet$$
$$\exists\, go : GeneralObject\ ;\ of : ObjectFlow\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet$$
$$of \in go.incomingEdges \land of.isOutputOf = mf.tr \land r''.isPlayedBy = go$$
$$\land\ r.tr \in if.targets \land \#(if.targets) = 1$$
$$\land\ r'' \in if.sources \land \#(if.sources) = 1$$

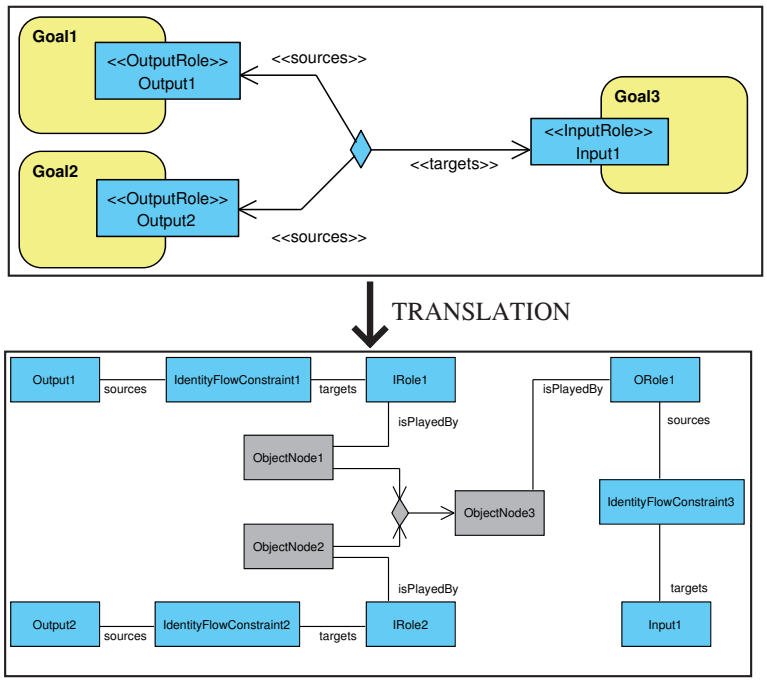


Figure 4.20: CG to AD-S: example of the Merge translation

Decision Flows The translation of DecisionFlows is similar to the one of MergeFlows.

$$| \; tr : DecisionFlow \longrightarrow Decision$$

$$\forall\, df : DecisionFlow \bullet \exists\, df' : Decision \bullet df.tr = df'$$
$$\forall\, df : DecisionFlow\ ;\ r : Role \mid r \in df.sources \bullet$$
$$\exists\, go : GeneralObject\ ;\ of : ObjectFlow\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet$$
$$of \in go.outgoingEdges \land of.isInputOf = df.tr \land r''.isPlayedBy = go$$
$$\land\ r.tr \in if.sources \land \#(if.sources) = 1$$
$$\land\ r'' \in if.targets \land \#(if.targets) = 1$$
$$\forall\, df : DecisionFlow\ ;\ r : Role \mid r \in df.sources \bullet$$
$$\exists\, go : GeneralObject\ ;\ of : ObjectFlow\ ;\ r'' : WFRole\ ;\ if : WFIdentityFlow \bullet$$
$$of \in go.incomingEdges \land of.isOutputOf = df.tr \land r''.isPlayedBy = go$$
$$\land\ r.tr \in if.targets \land \#(if.targets) = 1$$
$$\land\ r'' \in if.sources \land \#(if.sources) = 1$$

Control Flows The translation of ControlFlows is similar to the other FlowConstraints except that a Join node and a Fork node are created as middle layer and that WFControlFlow are used with sources and targets Roles.

$$\begin{array}{|l} trs : ControlFlow \longrightarrow WFRole \\ trt : ControlFlow \longrightarrow WFRole \end{array}$$

$$\begin{array}{l} \forall\, cf : ControlFlow \bullet \exists\, cf1 : WFRole\ ;\ j : Join\ ;\ cf2 : WFRole\ ;\ f : Fork \bullet \\ \quad cf.trs = cf1 \land cf1.isPlayedBy = j \\ \quad \land\ cf.trt = cf2 \land cf2.isPlayedBy = f \\ \forall\, cf : ControlFlow\ ;\ r : WFRole \mid r \in cf.sources \bullet \exists\, cf' : WFControlFlow \bullet \\ \quad r.tr \in cf'.sources \land cf.trt \in cf'.targets \\ \forall\, cf : ControlFlow\ ;\ r : Role \mid r \in cf.targets \bullet \exists\, cf' : WFControlFlow \bullet \\ \quad r.tr \in cf'.targets \land cf.trs \in cf'.sources \end{array}$$

Constraints

From the composition goal translation, we can note that all CG requirements are resolved in the end to only four new AD-S classes: the WFCompositionGoal class, the WFRole class, the WFIdentityFlow class and the WFControlFlow class.
Input and Output Roles of the CompositionGoal container are respectively the user required and user available messages.
Atomic goals lead through discovery to a set of SWSs choreographies. The AcceptEvent and SendEvents corresponding to input and output Roles of the matched atomic goal are added to the relation *isPlayedBy* of the appropriate WFRole.
A set of new configuration constraints is introduced to the AD-S model to ensure the computed workflow respects the semantics of the new AD-S objects (hence of the composition goal).

WFRoles The cardinality 1,1 of the relation isPlayedBy forces the composer to choose one and only one node from the relation's domain as being the node playing this role.

WFIdentityFlows The semantics are that messages (playing the source and target(s) WFRoles) are the same, hence the corresponding ActivityNodes must be connected by an object flow. Although such a flow must exist between source and target(s), it does not need to be a direct connection (i.e a single edge). Indeed, the path can contain *intermediate* nodes in between, as long as those intermediate nodes do not change the carried concept. This is achieved using the relation *intermediates*, with a restriction on control or mediation nodes
A identity flow target role's played by node must have an object flow coming from either

the source role's played by node, or an appropriate intermediate node.

$$
\begin{aligned}
&\forall\, n : \mathit{ActivityNode}\ ;\ r : \mathit{Role}\ ;\ c : \mathit{WFIdentityFlow}\ \mid \\
&\quad n = \mathit{isPlayedBy}(r) \land r \in \mathit{targets}(c)\ \bullet \\
&\quad \exists\, o : \mathit{ObjectFlow}\ ;\ n' : \mathit{ActivityNode}\ \bullet \\
&\quad n' \in c.\mathit{sources} \rightarrow \mathit{isPlayedBy} \\
&\qquad \lor\ (n' \in \mathit{intermediates}(c) \land (n' \in \mathit{ControlNode} \lor n' \in \mathit{Mediation})) \\
&\quad \land\ o \in \mathit{incomingEdges}(n) \\
&\quad \land\ \mathit{isOutputOf}(o) = n'
\end{aligned}
$$

Recursively, an intermediate node must have an object flow coming from either the constraint source role's playedBy node, or an appropriate intermediate node.

$$
\begin{aligned}
&\forall\, n : \mathit{ActivityNode}\ ;\ c : \mathit{WFIdentityFlow} \mid n \in \mathit{intermediates}(c)\ \bullet \\
&\quad \exists\, o : \mathit{ObjectFlow}\ ;\ n' : \mathit{ActivityNode}\ \bullet \\
&\quad n' \in c.\mathit{sources} \rightarrow \mathit{isPlayedBy} \\
&\qquad \lor\ (n' \in \mathit{intermediates}(c) \land (n' \in \mathit{ControlNode} \lor n' \in \mathit{Mediation})) \\
&\quad \land\ o \in \mathit{incomingEdges}(n) \\
&\quad \land\ \mathit{isOutputOf}(o) = n'
\end{aligned}
$$

ControlflowConstraint Similar to the IdentityFlow, the ControlFlow constraint is however easier to introduce. Indeed, there is no need for intermediate nodes as control flows are natively transitive. Therefore the constraint simply states that the targets role's playedBy node has an incoming control flow from the source role's playedBy node.

$$
\begin{aligned}
&\forall\, n : \mathit{ActivityNode}\ ;\ r : \mathit{Role}\ ;\ c : \mathit{WFControlFlow}\ \mid \\
&\quad n = \mathit{isPlayedBy}(r) \land r \in \mathit{targets}(c)\ \bullet \\
&\quad \exists\, o : \mathit{ControlFlow}\ ;\ n' : \mathit{ActivityNode}\ \bullet \\
&\quad n' \in c.\mathit{sources} \rightarrow \mathit{isPlayedBy} \\
&\quad \land\ o \in \mathit{incomingEdges}(n) \\
&\qquad \land\ \mathit{isOutputOf}(o) = n'
\end{aligned}
$$

The described translation allows to express any composition goal requirements in the workflow configuration model. As explained previously, the translation between the two models is implemented with a linear algorithm. However our description as an additional set of constraints offers the possibility to configure both in one unique model. Since such a process would raise important scalability issues with only little benefits (the possibility to "backtrack" on the CG-model), it is out of scope of our current researches.
Thanks to the two models and the translation, the whole composition process now allows to provide a partial composition goal with abstract requirements, that will be automatically completed and used as a request for the resulting composite workflow.

4.4 Towards automatic extraction of SWS descriptions from the composite workflow

In order to execute the composite SWS, we need to obtain its orchestration. In order to publish the composite SWS, we also need to obtain its capability and choreography. These descriptions are contained in the composite workflow, but it is not straightforward to obtain them. This Section proposes procedures for an automatic extraction. The process, illustrated in Figure 4.21, requires to abstract workflow information from the consumed choreographies so as to introduce it in the descriptions.

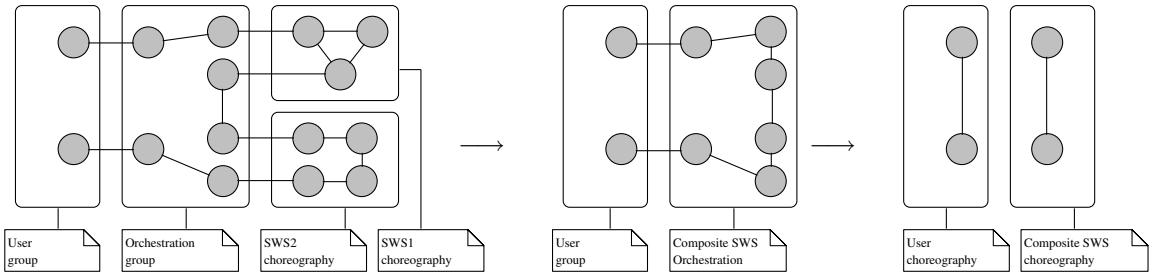


Figure 4.21: Schema of an automatic extraction of composite SWS descriptions

4.4.1 Obtaining the capability

The capability is easily obtained from the request. It is made of the user required output messages, and the set of input messages that are necessary in the composite workflow.

4.4.2 Obtaining the orchestration from the composite workflow

The composite workflow regroups the composite SWS choreographies, the user interaction, and the composed SWS internal elements. The orchestration should only contain the internal elements including the Accept/Send events which allow communications with the user and the composite SWSs.
Simply isolating the Orchestration group of the composite workflow would not yield a valid orchestration. On the one hand, we may loose the execution properties of the composite SWS. Figure 4.22 illustrates this. If we remove the choreography, the AcceptNode which waits for message B could never be fired depending on the rest of the orchestration workflow. In other words, the orchestration is no longer *compatible* with its consumed choreographies.
On the other hand, we wish to preserve control flow information that may be contained in the consumed choreographies. In Figure 4.22, we loose the information that receiving a message B takes place after having sent message A. We will therefore be unable to reflect this in the composite SWS choreography.

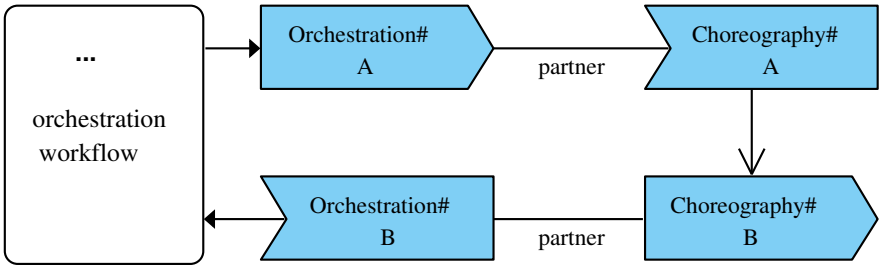


Figure 4.22: The problem of isolating the orchestration

Orchestration vs consumed service choreography

The required workflow compatibility between an orchestration and a consumed service choreography has been acknowledged in [16] and is known as *conformance*. Although there is work for checking the conformance of two communicating workflows [66, 107], to the best of our knowledge no existing work tackles the question of isolating a part of a composite workflow while preserving conformance between the separated parts.
Different levels of conformance can be considered. [107] differentiates fitness (whether the described behaviour is possible) from *appropriateness* (whether the described behaviour "overfits" or "underfits" the communication protocol).
In our context we are interested in conformance in the sense of a correctly described communication from the orchestration point of view. This conformance can be informally explained as follows: there exists at least one potential execution in which (1) each time a consumed choreography sends a message required by the orchestration, the orchestration is ready to receive it (2) each time the orchestration sends a message, the choreography is ready to receive it.

Reasoning about workflows executions

Workflows specified using AD-S have different potential executions. Indeed, the token flow is dependent on various *facts*: number of tokens provided by the user at a given SendEvent node, value of the incoming token and Guard conditions for a given Decision node, etc. Furthermore, the outcome of arbitrary Decisions in consumed SWSs cannot be computed thus introducing additional non-determinism.
In order to reason about one concrete execution of a workflow, it is possible to define a set of facts and conditions such that a deterministic execution (or simulation) of the workflow can be realized. This set contains statements such as "1 token t with value x is present in the user SendEvent se at the beginning" or "when token t arrives at the Decision node n, t will flow through the outgoing edge e".
We will thus refer to a concrete execution of a workflow using a set of facts, defined later as an *execution context*. Since it may not be possible at design-time to detect some inconsistent pairs of facts (think of arbitrary decisions), a current limitation of this work

is that an execution context does not necessarily reflect a possible execution. We will ignore these (seldom) cases in the following.

Extracting an orchestration with respect to conformance

We first introduce a set of definitions and propositions:

Definition 4.4.1 *execution context*
An execution context (D, w) *is defined by a set of facts and conditions* D *such that the workflow* w *can be deterministically executed (or simulated).*
We say that a node n *is* fired *under an execution context* (D, w) *if the facts* D *and the execution semantics of* w *yield the execution of* n *at a given moment of the execution.*

Proposition 4.4.1 *For all node* $n \in cw$ *(cw being a configured composite workflow) such that* $active(n)$ *is true, there exists an execution context (D,cw) firing* n*.*

Proof 4.4.1 *we propose a proof by induction that stems for the AD-S execution semantics and model constraints.*
(1) For all node $n \in cw$ *such that* n *is a SendEvent in the user choreography, there is a fact* f *such that* n *is fired (the user provides the information).*
(2) For all node $n \in cw$ *such that* n *is an Initial node, it is fired at the beginning for any execution context (execution semantics of an Initial node).*
(3) For all other nodes $n \in cw$ *such that* $active(n)$ *is true, the constraints ensure that a set* $inputs(n)$ *of incoming edges (and, in the case of Action nodes, InputPins, and, in the case of AcceptEvents, partners) are active. The property is that, if every element of this set* $inputs(n)$ *holds a token, then the node* n *is fired according to its execution semantics. We thus consider any potential element* $el \in inputs(n)$*.*
(4) If el *is an edge, it is output of a single node* n'*. If* n' *is not a Decision node, then* el *receives a token whenever* n' *is fired and we may resursively apply statement (1) or (2) or (3). If* n' *is a decision node, there is a condition* d *such that* el *receives a token when* n' *is fired and we can apply statement (3) by adding this condition* d *to the execution context.*
(5) If el *is an InputPin, we may resursively apply statement (3).*
(6) if el *is a SendEvent, we may recursively apply statement (3).*
(7) As the number of nodes and edges in cw is finite and cw is acyclic, the induction on every workflow path will stop. Furthermore, every workflow path stops at either at an initialNode or at a user SendEvent, as all other nodes have incoming edges and/or input pins, and/or a partner.
As a consequence, there exists an execution context (D, cw) *such that* n *is fired.*

Definition 4.4.2 *Duals*
We call dual node *of an Event its partner Event in a composite workflow (notation* $dual(e) = e'$*). The AD-S model constraints ensure that this relation is bijective i.e* $dual(e) = e' \leftrightarrow dual(e') = e$*.*
We define an activity group g_1 *as a dual workflow of* g_2 *(notation* $g_1 \in duals(g_2)$*) if there exists an event* $e \in g_1$ *and an event* $e' \in g_2$ *such that* $dual(e) = e'$*.*

Following our informal description of conformance, we give it a context-based definition:

Definition 4.4.3 *basic conformance*
A workflow w_1 is basically conformant (notation $w_1 \xrightarrow{c} w_2$) with a dual workflow $w_2 \in duals(w_1)$ if for each event $e \in w_1$, there exists an event $e' \in w_2$ such that:
(1) $e' = dual(e)$,
(2) if e is an AcceptEvent, there exists an execution context (D, w_1) firing e such that e' is fired in the execution context (D, w_2),
(3) if e is a SendEvent, there exists execution context (D, w_1) firing e such that e' is fired in the execution context (D, w_2).

Finally we transpose this definition to our composite workflows:

Definition 4.4.4 *global conformance of composite workflows*
A composite workflow cw is globally conformant if for each event $e \in o$ (o being the orchestration group), there exists an event $e' \in C$ (C being the set of consumed choreographies groups) such that:
(1) $e' = dual(e)$,
(2) if e is an AcceptEvent, there exists an execution context (D, cw) firing e such that e' is also fired,
(3) if e is a SendEvent, there exists execution context (D, cw) firing e such that e' is also fired.

Proposition 4.4.2 *composite workflows that are solutions of the AD-S model are globally conformant*

Proof 4.4.2 *(1) each event $e \in o$ is active, and thus has an active partner $e' = dual(e) \in C$.*
(2-3) With Proposition 4.4.1, we know there exists an execution context (D, cw) firing e, and an execution context (D', cw) firing e'. As a result, the execution context $(D \cup D', cw)$ fires e and e'.

We are interested in preserving this global conformance when isolating the orchestration group from the rest of the composite workflow:

Proposition 4.4.3 *An orchestration group o,* conformant *with its set of consumed choreographies C, can be isolated from a globally conformant composite workflow cw if for each choreography group $c_i \in cw$, there is a conformance $o \xrightarrow{c} c_i$.*

Proof 4.4.3 *choreographies never communicate directly between each other. Therefore their execution context is independent and a pair-wise conformance $o \xrightarrow{c} c_i$ ensures a conformance of the isolated orchestration with its set of consumed choreographies C.*

When isolating the orchestration, the tokens no longer flow along the partner relations. As a consequence, the firing of a choreography's SendEvent does not yield a token in its partner (orchestration's) AcceptEvent. If the AcceptEvent had no other incoming

edges, it may never be fired (recall Figure 4.22).
A simple workaround would be to add a connected initial node to all orchestration's AcceptEvents. However, with respect to conformance appropriateness, it is better suited to fire the AcceptEvent only under the same execution context as its corresponding SendEvent, thus preventing unnecessary activations. Another argument is that a client choreography will, in a further step, be computed from the orchestration. Therefore we should preserve in the orchestration as much execution information as possible.
We propose a method for simulating the required workflow information into the orchestration through the notion of *dual firing paths*.

Definition 4.4.5 *firing path*
A workflow path $p(s,t)$ is a set of connected nodes and edges which constitute a path from a source node s to a target node t.
A firing path $p((D,w),s,t)$ is a workflow path such that the execution of s participates *to the firing of t under the execution context (D,w).*
We say $p'((D,w_2),s',t')$ is a dual firing path of $p((D',w_1),s,t)$, notation $p' = dual(p)$, if $D' = D$, $dual(s) = s'$ and $dual(t) = t'$.

We can first make an assumption on the context of our workflows communications: (1) the interaction with a choreography starts at the initiative of the orchestration (2) the corresponding choreography is ready to receive the initial(s) message(s) . This is rather obvious as (1) a consumed SWS does not know anything of its client before the client engaged communication (and thus would not be able to send it a message) (2) the choreography client needs a (set of) messages it can engage communication with. Although this assumption is not mandatory in the following, the resulting procedure and proofs will be easier, thanks to the following consequences:

Definition 4.4.6 *For any choreography c, there is a non-empty set of AcceptEvents c_{IES} (IES stands for Initial Event Set) fired in any execution context.*

Proposition 4.4.4 *For any AcceptEvent $ae \in o$ with $dual(ae) \in c$, there is a SendEvent $se \in o$ such that there exists a workflow path $p(dual(se), dual(ae))$ and $dual(se) \in c_{IES}$.*

Proof 4.4.4 *If there exists an AcceptEvent $ae \in o$ with $dual(ae) \in c$, the orchestration had already engaged communication. Therefore there exists a SendEvent $se \in o$ with $dual(se) \in c_{IES}$, and $dual(ae)$ is connected to $dual(se)$.*

We now return to the conformance of our isolated orchestrations:

Proposition 4.4.5 *An orchestration o, isolated from a globally conformant composite workflow cw, is conformant ($o \xrightarrow{c} c$) with a dual choreography $c \in duals(o)$ if:*
for each pair of events $e_1, e_2 \in o$ such that $e_1' = dual(e_1) \in c$, and $e_2' = dual(e_2) \in c$, and for all firing paths $p'((D,c),e_1',e_2')$; there exists a dual firing path $p = dual(p')$.

Proof 4.4.5 *As cw is globally conformant we have (1)*[4] *each event $e \in o$ is active, and thus has an active partner $e' = dual(e) \in C$.*
(3a) for all $e \in o$ such that $dual(e) \in c_{IES}$, e' is fired in any execution context, and thus any execution context (D, o) firing e also fires $dual(e)$.
(2-3b) for all $e \in o$ such that $dual(e) \notin c_{IES}$, there is at least one firing path $p((D, o), e_s, e)$ such that $dual(e_s) \in c_{IES}$ and $p = dual(p'((D, c), dual(e_s), dual(e))$. Therefore there exists an execution context firing e that also fires $dual(e)$.

These propositions induce an automatic procedure for obtaining a globally conformant orchestration: a pair-wise basic conformance must be established between the orchestration and the consumed choreographies. This basic conformance can be obtained, with a certain degree of appropriateness that preserves the control flow information from the consumed choreographies, by duplicating the control flow of the choreographies related to pairs of partner events in the orchestration and this can be achieved by generating dual firing paths.

Simplifying the choreographies

Before we generate the dual firing paths, we first simplify the participant choreographies based on the following observations:

- (1) only the control-flow of a composite choreography is relevant to conformance. Indeed, only data-flow combined with a deterministic decision can have a computable effect on the control-flow. As we consider decisions in a choreography as non-deterministic (the external choice workflow pattern), we do not lose information by restricting the edges to control-flow.
- (2) only the active paths between events having a dual in the orchestration (i.e active events) are relevant to conformance.

We thus apply the following rules to choreographies:

- (1a) A GeneralObjectNode is replaced with a Fork
- (1b) An ActionNode's InputPins are replaced with a Join. An ActionNode's OutputPins are replaced with a Fork. A ControlFlow is placed from the created Join to the created Fork
- (2a) A path between an InitialNode and an Event is removed
- (2b) A path between an Event and a FinalNode is removed

[4] Each (number) refers to the definition of conformance.

```
procedure simplify
  do
    aRuleCanBeApplied:=applyAnyRule1
  while (aRuleCanBeApplied)
  do
    aRuleCanBeApplied:=applyAnyRule2
  while (aRuleCanBeApplied)
```

Table 4.1: The procedure **simplify**

- (2c) Any edge or node for which active is false is removed. There is however an exception to this rule: if the non-active path is a path from an active decision to an unactive SendEvent. This will allow us to produce a basic exception handling as discussed later
- (2d) a node with a single incoming edge and a single outgoing edge is removed and replaced by a single edge between source and target.

Proposition 4.4.6 *Active firing paths between events of the choreography are preserved by the simplification.*

Proof 4.4.6 *(1a-1b) The consuming/producing of tokens is preserved by each rule: the execution semantics of ObjectNodes (or ActionNodes) are the same as a join on incoming edges (or input pins) and a fork on outgoing edges (or output pins).*
(2a-2b) Firing paths between events do not include either paths from initial nodes nor paths to final nodes: initial nodes do not have incoming edges, and final nodes do no have outgoing edges.
(2c) Active paths are not removed.
(2d) This is obviously equivalent to an edge from source to target.

We use a simple fixed-point algorithm described in Table 4.1 to apply the simplification rules. The function ApplyAnyRule1 applies rules (1a-1b) whereas ApplyAnyRule2 applies rules (2a-2d).

Proposition 4.4.7 *The procedure* simplify *terminates.*

Proof 4.4.7 applyAnyRule *returns true if any rule has been applied. If a rule is applied, either the number of edges has been reduced, or an ActionNode has been removed, or an ObjectNode has been removed. As the number of edges, ActionNodes and ObjectNodes is finite, applyAnyRule1 and applyAnyRule2 will eventually return false and the procedure* simplify *terminates.*

An example of a choreography simplification is presented in Figure 4.23.

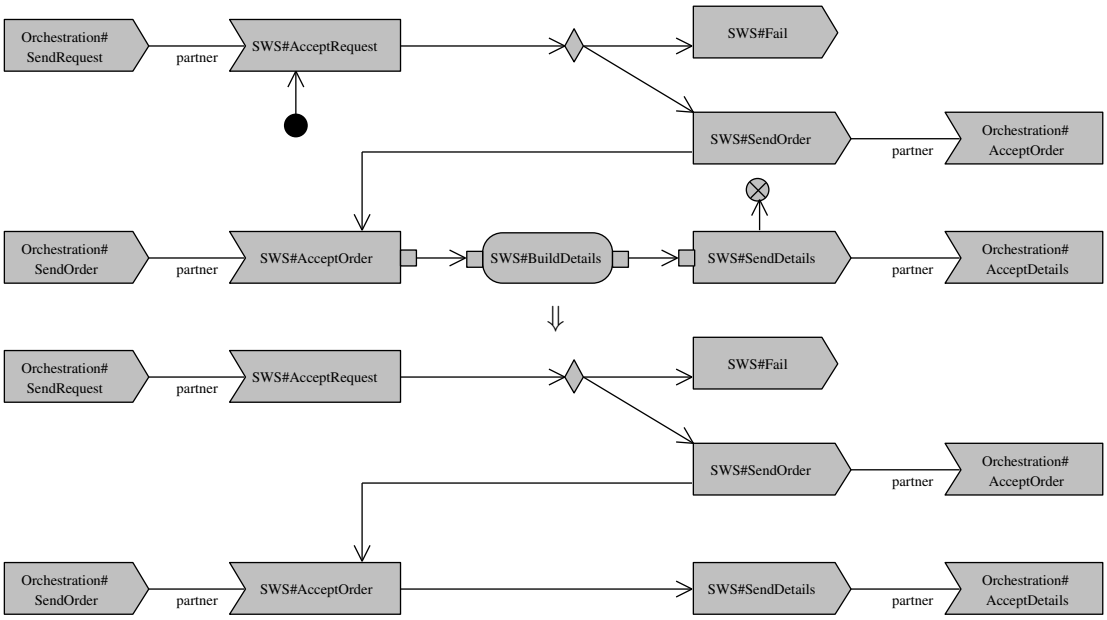


Figure 4.23: AD-S: simplifying a choreography

Generating dual firing paths

We now create the dual firing paths in the orchestration corresponding to active firing paths (between duals of the orchestration events) in the choreography. We use a recursive algorithm presented in Table 4.2. The main idea is to generate a dual edge for each edge of the firing path: for each Event e in the Choreography having a dual node e' in the Orchestration, we create a dual edge e'_i going out from e' for each outgoing edge e_i of e. If e_i's target $target(e_i)$ has a dual $dual(target(e_i))$, the latter becomes the target of e'_i. Otherwise if $target(e_i)$ is another type of node, we create a dual node in the orchestration and recursively treat outgoing edges of $target(e_i)$.
The creation of a dual node should preserve the execution semantics of the original node. It is based on the following rules:

- If $target(e_i)$ is a SendEvent without dual, it corresponds to an unexpected behaviour. As a basic exception handling facility, we create a dual AcceptEvent which leads to the end of the orchestration's execution.
- if $target(e_i)$ is a decision node, the orchestration cannot infer the outcome during execution. This is the workflow pattern External choice we discussed before. As we already argued, this is modelled in UML2AD with an InterruptibleRegion. We thus create a Fork and an InterruptibleRegion which will contain the duals of the Decision's targets (hence the targets of the newly created Fork). One outgoing

```
procedure generateDuals
   for each event e in the choreography
      for each outgoingEdge e_i do
         addAbstractedEdge(dual(e),target(e_i))
   create InterruptingEdges for abstracted InterruptibleRegions
   return

function edge addAbstractedEdge(source,target)
   if((isAcceptEvent(target) or isSendEvent(target))
         and target.partner in orchestration)
      return createEdge(source,target.partner)
   else if(target in abstractedNodes)
      return createEdge(source,abstractedNodes(target))
   else
      return createEdge(source,addAbstractedNode(target))

function node addAbstractedNode(originalNode)
   node:=createAbstractedConstruct()
   return node
```

Table 4.2: The procedure **generateDuals** with its functions **addAbstractedEdge** and **addAbstractedNode**.

edge of each of the Fork's targets will be made interruptingEdge of the InterruptibleRegion.

- For all other types preserved by the simplification (Fork,Join,Merge), a dual node of the same type is created.

Proposition 4.4.8 *The procedure* generateDuals *terminates*

Proof 4.4.8 *The choreography is acyclic, the procedure never visits twice the same node or edge, and the number of edges and nodes is finite.*

Proposition 4.4.9 *The procedure* generateDuals *correctly generates in the orchestration o the dual firing paths of all firing paths in the choreography c.*

Proof 4.4.9 *(1) For each edge(source,target) belonging to a firing path in c, a dual edge(dual(source),dual(target)) is created.*
(2) If the source node (or target) is an Event in c, dual(source) (or dual(target)) is the partner Event in o. Otherwise, it is an abstracted node in o preserving the execution semantics of the firing path.
(2) For each node $n \in c$, there can be at most one abstracted node dual(n) created.

An example of the automatic generation of dual firing paths is presented in Figure 4.24. It immediatly follows the simplification presented in Figure 4.23. The full example is taken from the NMPC-bundle scenario and only shows one choreography and the corresponding orchestration Events.

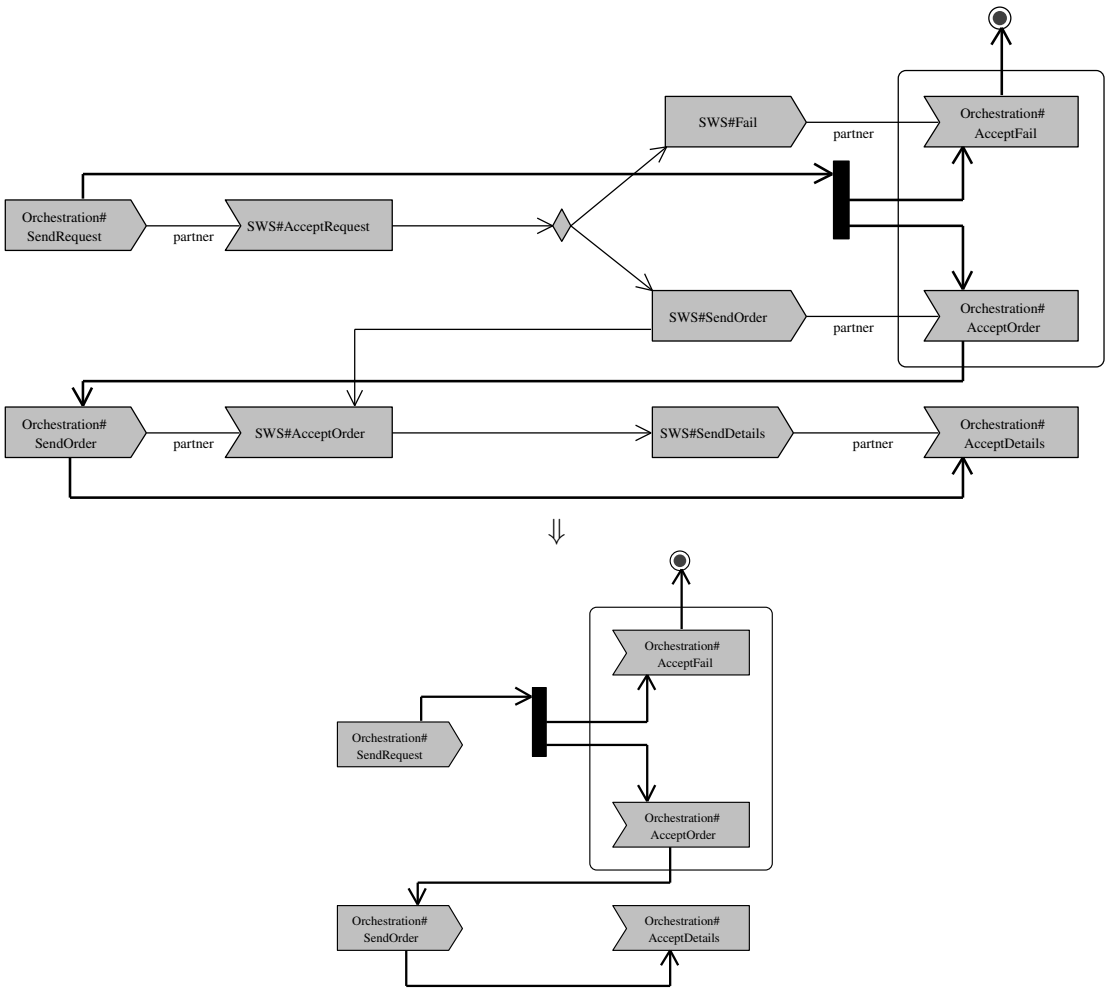


Figure 4.24: AD-S: generated dual firing paths and removal of the choreography

4.4.3 Obtaining the choreography from the orchestration

A similar procedure can be applied to extract a client choreography from the orchestration. However, we do not detail the process.
The main idea is that the orchestration is simplified to its control flow related to the client interaction. The simplification is close to the one previously described, with the

difference that Events having another partner than the user can be removed. The result is the composite SWS choreography. A similar generation of dual firing paths is then applied to the user Events to obtain the user choreography.

4.5 Composer implementation and integration in the European project DIP

4.5.1 Integration in DIP's Framework

The chosen framework in DIP is mostly based on the WSML/WSMO/WSMX architecture (WSML is the language; WSMO is an ontology for SWSs, goals, mediators and ontologies; WSMX is an execution environment). The DIP framework also makes use of the IRS-III (Internet Reasoning Service) infrastructure which has a direct support for WSMO.
A set of tools has been developed within the DIP project. In particular, the composer can use:

- two discovery engines: the IRS-III discovery engine and the WSM discovery engine,
- the WSMX execution environment as an orchestration engine,
- WSMO studio: a java environment for modeling SWSs, accessing SWSs registries, and invoking the different DIP tools.

The interaction with the tools using different formalisms is obtained through the integration of our AD-S language in a shared 3-layer description of SWSs choreographies and orchestrations.

Integration in a 3-layer Behavioural Models for Semantic Web Services

The 3 levels consist of the AD-S model, a WSMO-based evolution of the Cashew-S ontology, and ontologized ASMs (an ontology of Abstract State Machines is the WSMO proposal for choreographies and orchestrations descriptions). Figure 4.25 shows the 3 layers and their main characteristics. [11, 37, 61] are the related project's deliverables.
The WSML grammar has been extended so that each layer can be expressed in it. Figure 4.26 is a screenshot of WSMO Studio showing a 3-layer description of a choreography in WSML. The AD-S/WSML grammar is given in Annex II.
Finally a translation from AD-S to Cashew-S and from Cashew-S to ontologized ASMs is given in [61, 78].

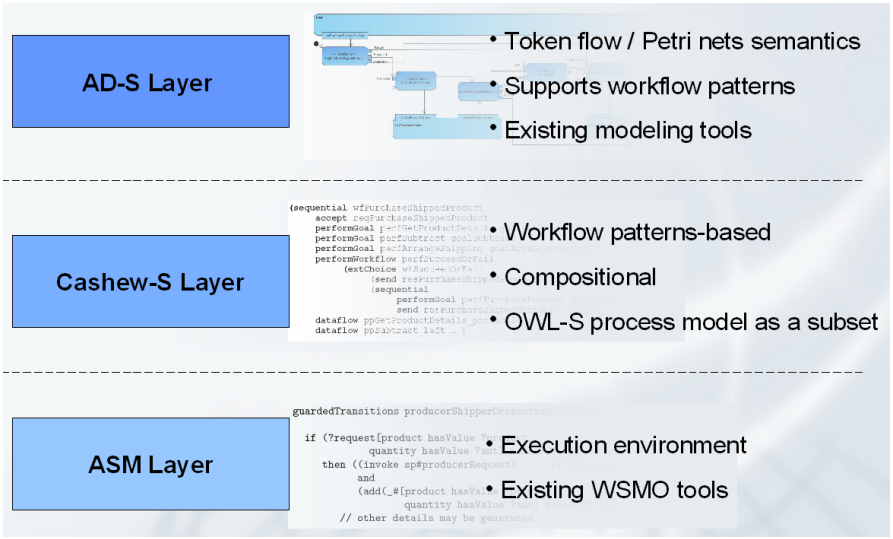


Figure 4.25: 3-layer behavioural models for Semantic Web Services

4.5.2 Composer implementation

The composer has been developed in two versions: a stand-alone program and a WSMO studio plugin for DIP integration. Both are implemented using the Java language. Specifications and prototypes have been presented in the project's deliverables [1, 2].

Components

The implemented composer is made of several components:

- a java representation (JCG) of composition goals and a XML-parser for importing requests
- a java representation (JChorch) of AD-S choreographies and orchestrations and a XML-parser for importing discovered choreographies
- a configurator (we used ILOG's JConfigurator)
- a translation between JCG and JConfigurator's CG model
- a translation between JChorch and JConfigurator's AD-S model
- a linear algorithm to translate CG objects to AD-S objects and constraints
- a fixed-point algorithm and a recursive algorithm to extract conformant orchestrations from composite workflows

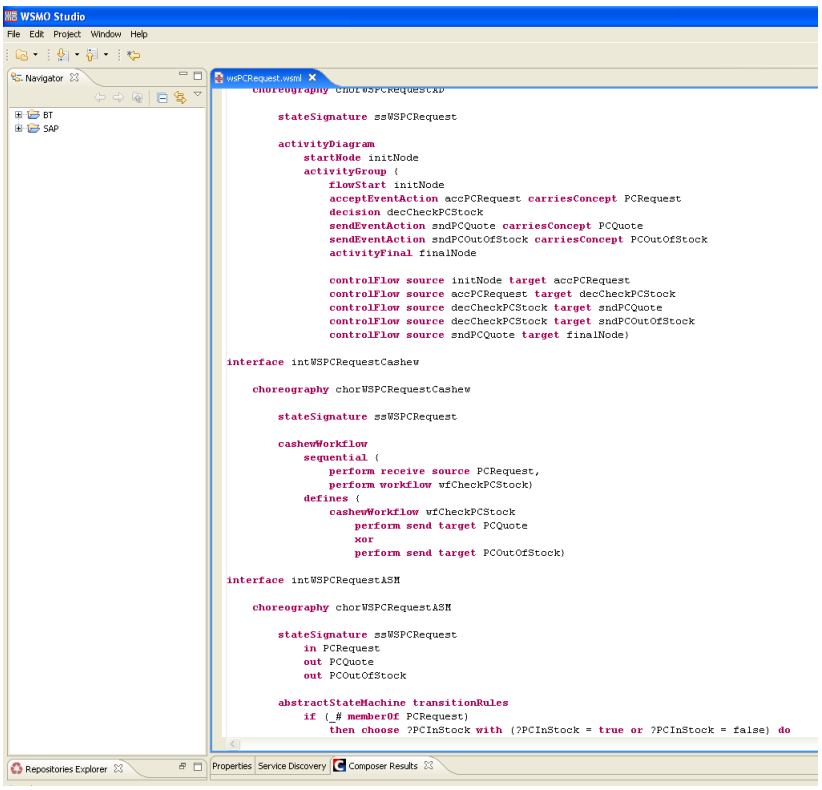


Figure 4.26: WSMO Studio screenshot with the 3-layer choreography in WSML

- an import/export facility for interaction with the other tools (for instance exporting orchestrations to the AD-S WSML grammar)

Using JConfigurator for configuration tasks

We implemented the CG-model and the AD-S model in ILOG's JConfigurator, based on a modular translation from Z-based COMs to JConfigurator's COMs. Fragments of the code for JConfigurator's AD-S model are given in Annex I.

4.6 Experimental results

We provide experiments on four scenarios. The scenarios have been integrated in a full SWS framework, from composition (out of a full composition goal) to execution. We do not consider in those experiments the possibility of automatically configuring the composition goal but we accept them with any allowed construct as process-level requests. The producer-shipper-a and producer-shipper-bank scenarios are taken from [106]. We do not make use of composition goals. In both scenarios, an aggregated order confirmation is required from the user and implies interleaving SWSs execution. These scenarios

scenario	available SWS	candidate SWS	composition time (seconds)	choice points	selected SWS
PS-a	2	n/a	1.33	90	2
	10	n/a	3.36	96	2
	20	n/a	6.77	116	2
PSBank	3	n/a	1.68	102	3
PS-b	10	2	4.17	84	2
	10	3	21.25	159	2
	10	4	68.10	269	2
NMPC	20	4	3.52	51	4

Table 4.3: Configuration-based composer experiments

can serve as a comparison with the STS composer proposed in [83], which is one of the most efficient available composers.
The producer-shipper-b scenario is a refinement proposed by a DIP use-case partner (SAP) where the involved choreographies have a more complex behaviour. Their choreography describes a communication with 8 request/response patterns between the shipper and the producer. The request here is a simple composition goal restricting the SWS library but without any value or flow constraints.
The NMPC-bundle scenario, extensively presented through this chapter, has been joint developed with another DIP use-case partner (British Telecom).
Table 4.3 shows results. Experiments were conducted on a Pentium IV 2.8GHZ with 512MB of RAM. The second column gives the number of existing SWS whereas the third column gives the number of candidate SWSs returned by discovery on the basis of the required atomic goals. The composition time includes linear translation of the composition goal, configuration of the composite workflow and extraction of a conformant orchestration. The number of choice points during JConfigurator's solving is given in the fourth column.
No special heuristics have been used during the configuration task apart from JConfigurator's default ones.
On simple scenarios like PS-a and PSBank, when the number of available SWS is restricted to the necessary ones, the composer is efficient. However increasing the available services (the added services are copies of the original ones) increases composition times above what we may expect for an "on-the-fly" composer.
The impact of composition goals on scenarios PS-b and NMPC is obvious: thanks to discovery interaction, the number of candidate SWS is reduced to a set of potentially useful services, and the configuration search space is thus limited. If we increase the total number of available SWS with services which do not match the atomic goals, the composition is unaffected.
However the PS-b usecase shows that with complex choreographies, increasing the number of services (again, we provided copies) has a large effect on composition times. This

	CompLevel		NAC	WP			DR	SWS	Req.	Disc.	Tool
	goal	process		(a)	(b)	(c)					
	+	+	+	+	+	-	+/-	+	+	+	+

Table 4.4: Configuration-based composer features (+) = supported, (-) = not supported, (+/-) = partially supported

is due to the combinatorial explosion in the presence of an important number of nodes and edges.
In the NMPC-bundle scenario, we can highlight the advantages of complex composition goals. Here we provided many services selling a network connection, but only one of them offers ADSL connection. Thanks to the propagation of UnaryValueConstraints (such as the NetworkDetails' connectionType) to an atomic goal's roles, we reduce the search space for the composer. Indeed, it restricts the candidate SWSs (in the current example only SWS offering ADSL connections will be discovered).
Finally, the ability to (manually, semi-automatically or automatically) fine-tune the composition request very precisely creates a set of workflow constraints. These additional constraints reduce the configuration space and discard unwanted orchestrations.

Comparison to existing composers

Table 4.4 shows the level of composition addressed by our configuration-based composer based on the features introduced in the state of the art.
There are two main limitations of our approach. Firstly, we do not offer support for workflow patterns such as loops or multiple instances. Although UML2AD supports them through *structured activities*, we have not yet investigated their addition to our COM.
Secondly, we only partially reason on data ontologies (inheritance and composite concepts) and this is only implemented at the goal-level.
If we consider efficiency in terms of computation times, the comparison is restricted as seldom researches have published experimental results on concrete scenarios. However we can compare two of our scenarios with the results obtained with the STS approach [83]. This is presented in Table 4.5.
Although we found that the configuration-based composer suffers from a combinatorial explosion, it can easily compete with one of the most efficient known approaches. These results must however be analysed carefully since the representation formalisms and experimental framework differ greatly.

Perspectives

Perspectives for ontology reasoning Data reasoning is realized on a defined metamodel of ontologies. This induces a modelling effort for more advanced reasoning, for instance on concepts attributes. If we were to use the direct model of an ontology,

Scenario	STS time	Config time
PS-a	9.4	1.33
PSBank	75.0	1.68

Table 4.5: SWS composition experimental comparison, times in seconds

instead of its meta-model, the search should only refine the concepts upon necessity. However actual configurators will try to instantiate any object participating in the solution whereas at design-time it is not be expected to decide on execution values of concepts unless a restriction is necessary.
In ILOG's configuration tool, a step has been taken in this direction with the modelling concept of *class relations*: the target is classified but not configured as an object. The generalization of this behaviour would be a significant change in configuration's fundamentals, which has to see with dynamic meta-model reasoning.

Perspectives for workflows executability The executability of the configured composite workflows can be improved in various ways. For instance, some executions require multiple activation of the same node. It is envisioned to mark the number of tokens that need to traverse nodes and edges. It would then be possible to propagate this to the number of times the user needs to send a message, or simply discard those workflows.
Another perspective is to introduce ontology reasoning and tokens value restrictions in the workflow model. It would, for instance, allow to detect additional inconsistencies at design-time.

4.7 Conclusion

This research describes how constraint-based configuration can be used for symbolic reasoning in the context of SWSs composition. The proposed approach deals with both goal-level and process-level composition. We also provide a method for extracting the related SWS descriptions where original issues are raised and given a context-based solution. A formal and reproducible composition process is described and experimentally validated in a concrete SWS framework.
The composer does not cover a certain number of functionalities that could be studied in future work such as compensation, more complex data reasoning or support of additional workflow patterns. In this sense, interesting perspectives are offered by the expressive power of the chosen workflow language (UML2AD) and the reasoning capabilities brought by configuration.
However the proposed specification already describes a composer which can easily compete with existing approaches. From the expressive power point of view, no other composer that we know of allows such freedom in the declaration of requests while supporting complex workflow descriptions. Although computational comparisons are restricted in

this field, we experimentally showed its efficiency on several scenarios.
Through this application example, we illustrated that configuration is a viable option in modern AI problems of first-order theories. One of the advantages offered by COMs is to allow for specifications and solving at an abstract level close to the associated knowledge domain. On the other hand we revealed computational issues which stem from the combinatorial nature of enumerative search, particularly prominent in the web context. The next chapters will concentrate on methods for improving configuration solvers.

Chapter 5

Isomorphisms Rejection for Configuration

In this chapter we propose a method to increase efficiency of finite models enumerative search. An inherent difficulty in solving configuration problems with complete algorithms is the existence of many structural isomorphisms. This issue of considerable importance attracted little research interest despite its broad applicability to configuration.
This research follows the work done in [44, 51] which presents a definition of canonicity for configuration trees and an enumeration algorithm that can be used for configuration problems involving solely composition constraints. Their main result is that the algorithm can backtrack as soon as a non canonical tree is constructed. We generalize it to generate arbitrary configuration structures in a complete and irredundant way.
We first define the structural sub-problem of configuration and in particular discuss the relation of isomorphisms rejection with symmetry breaking in classical CSPs. We recall the main results obtained in canonical tree generation. We then extend the properties and algorithms to the general configuration case, where configurations can be represented as directed acyclic graphs. Testing (weak) canonicity remains polynomial, and significantly impacts the behaviour of enumeration. Finally, we provide theoretical and experimental results on a range of problems.

5.1 Configuration structural sub-problems and isomorphisms

Configuration problems generally exhibit solutions having a prominent structural component, due to the presence of numerous composition relations. Many isomorphisms exist among the structural part of configuration solutions. Several approaches were experimented to tackle these symmetries, mostly by reasoning at a single level preventing redundant connections of interchangeable objects during search, or substituting the connection of actual objects by counting them according to their target types [65]. One may observe that general objects (i.e without predefined attributes or relations) are trivially

interchangeable as long as they have not been used in the configuration and share the same type.
The following simple example of a configuration problem will allow us to illustrate important notions throughout this paper. The problem is to configure a network of computers (C) and printers (P) (as illustrated in Figure 5.1). The network involves up to three computers, each of which being connected to at most two printers. Conversely, each printer must be connected to at least one and at most three computers. Besides this, we have two global constraints: there is only one network, and there are only two printers available. In a real problem, computers and printers could have specific attributes that would be instantiated while obeying other constraints. The impact of this on isomorphisms can be left aside as we solely focus on structural constraints (once a structure has been chosen for a configuration problem, it amounts to a classical CSP, to which all known symmetry breaking procedures for that case apply). Solutions to configuration

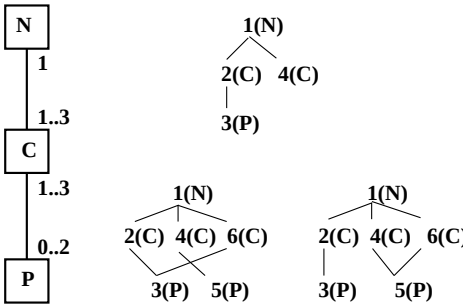


Figure 5.1: A network connection problem. On the left, the model for the components types (network, computers and printers) and their relations. On the right, 3 examples of possible structures. The two structures at the bottom are isomorphic and therefore represent equivalent solutions.

problems involve interconnected objects, as illustrated in Figure 5.1, where the existence of structural isomorphisms is obvious. From general configuration problems, we isolate configuration sub-problems called *structural problems* that are built from the binary relations, the related types and the structural constraints alone. The objective of the current section is to study their isomorphisms. For simplicity, we abstract from any configuration formalism, and consider a totally ordered set O of objects (we normally use $O = \{1, 2, \ldots\}$), a totally ordered set T_C of type symbols (unary relations) and a totally ordered set R_C of binary relation symbols. Given a binary *injective* relation $R \in R_C$, and the tuple $(x, y) \in R$, we use the notation $y = R(x)$ for simplicity when possible.

Definition 5.1.1 (syntax) *A* structural problem*, is a tuple* (t, T_C, R_C, C)*, where* $t \in T_C$ *is the root configuration type, and* C *is a set of structural constraints applied to the elements of* T_C *and* R_C*.*

In the network problem of Figure 5.1, we have t=`N`, T_C ={`N`, `C`, `P`}, R_C = {`(N,C)`, `(C,P)`}

and C is the set of structural constraints which enforce the minimum and maximum number of objects that can be connected for each binary relation.

Definition 5.1.2 (semantics) *An* instance *of a structural problem* (t, T_C, R_C, C) *is an interpretation I of* t *and of the elements of* T_C *and* R_C*, over the set* O *of objects. If an interpretation satisfies the constraints in* C*, it is a* solution *of the structural problem.*

We equally use the terms *structure* or *configuration* to denote a solution to a structural problem.
A configuration structure can be represented using a vertex-colored directed acyclic graph (DAG) G=(t,X,E,L) with X $\subset O$, E $\subset O \times O$ and L $\subset O \times T_C$. The symbol t is the root type, X the vertex set, E the edge set and L is the function which associates each vertex to a type. As an example, the upper solution of Figure 5.1 can be represented by the quadruple (`N`, {1,2,3,4}, {(1,2), (2,3), (1,4)}, {(1,`N`), (2, `C`), (3,`P`), (4, `C`)}).

Definition 5.1.3 (Isomorphic configurations) *Two configurations G=(t, X, E, L) and G'=(t', X', E', L') are isomorphic iff t=t', L=L' and there exists a one-to-one mapping* σ *between X and X' such that* $\forall$ *x,y* $\in$ *X,* $(x,y) \in E \Leftrightarrow (\sigma(x), \sigma(y)) \in E'$ *and* $\forall$ $(x,l) \in L$, $(\sigma(x),l) \in L'$.

For instance the two solutions at the bottom of Figure 5.1 are isomorphic since σ =((1,1), (2,4), (3,5), (4,2), (5,3), (6,6)) is a one-to-one mapping satisfying the definition criteria. Testing whether two graphs are isomorphic is an NP problem until today unclassified as either NP-complete or polynomial. The corresponding *graph isomorphism complete* class holds all the problems having similar complexity[1]. For several categories of graphs, like trees of course but also graphs having a bounded vertex degree, this isomorphism test is polynomial [63]. The "graph iso" problem is known however as weakly exponential, and there exist practically efficient algorithms for solving it, the most efficient one being Nauty [71]. This being said, we must emphasize now that Nauty cannot be used in our situation. The reason is that we must maintain the property that all canonical structures can be obtained from at least one smaller solution itself being canonical. Using Nauty from within an arbitrary graph enumeration procedure yields a generate and test algorithm: the portions of the search space that can be explored by adding to a non canonical structure must still be generated, in case they would contain canonical representatives which cannot be obtained differently. This situation will be explained in more detail in a forthcoming section.
An *isomorphism class* represents a set of isomorphic graphs. All the graphs from a given isomorphism class are equivalent, therefore a graph generation procedure should ideally generate only one *canonical* representative per class. This is of crucial importance since the size of an isomorphism class containing graphs with n vertices can be up to $n!$ (the number of permutations on the vertex set that actually create a different graph). Isomorphism classes are huge in size in most cases because, counter-intuitively, the less symmetrical a graph is, the more isomorphic graphs it has. This means that when

[1]For instance, the vertex-colored DAG isomorphism problem.

current configurators (which do not avoid isomorphisms or in a very restricted way) generate a solution, partial or complete, they also generate an often exponential number of isomorphic solutions.
Most graph generating procedures rely upon the central operation of adding an edge to an existing graph (operation called *unit extension*). Starting from a given graph, all its possible unit extensions yield a new set of graphs (which may or not be matching the model constraints). Having an efficient canonicity test is of little help for generating canonical graphs. Testing graphs for canonicity can be used to reject redundant solutions, but in so doing one has to explore the entire search space. The main intuition of the work presented here is as follows. It is obvious that unit extension naturally induces a backtrack search procedure for enumerating graphs, as new edges can be introduced that connect existing vertices, or an existing vertex to a newly introduced one. We wish to allow backtracking to occur as soon as a non canonical graph is produced while remaining complete wrt. canonical graphs. To achieve this, the canonicity criterion must be defined in such a way that for each canonical graph there exists at least one canonical subgraph resulting from the removal of one of its edges. We call this property the *canonical retractability property*. This condition is necessary (but not sufficient, since the enumeration procedure interferes with it as will be seen later) to allow for backtracking as soon as a non-canonical graph is detected during the search. Indeed if there exists a canonical graph not obtainable via extension of a canonical subgraph, the extension of a non-canonical graph will be needed to reach it. Such a canonicity criterion is not trivial to find, and most known canonicity tests, Nauty inclusive, do not respect it. There exist isomorphism-free graph generation procedures that impose conditions on the canonicity test, as for instance the *orderly algorithms* from [87] which however do not propose an efficient canonicity test. To the best of our knowledge, such an efficient test has not yet been found in the general case (if ever one exists). Specialized and efficient procedures for generating canonical graphs exist for trees, for cubic graphs [14] and more generally, for graphs having hereditary properties[2] [70]. Configuration problems unfortunately do not comply with these restrictions, which led us to develop specific procedures. In order to achieve this, we have based our research upon existing work around configuration problems.

5.1.1 Related work in CSP and configuration

Configuration problems versus CSP

Graph abstractions can be used to compare constraint formalisms (as in [92]) and allow for understanding the relative properties of configuration and CSPs. As presented in Chapter 2, a CSP is defined as a triple $< X, D, C >$ where X denotes a set of variables, D denotes a set of domains for these variables, and C denotes a set of constraints that control valid variable assignments. When the domain of variables is allowed to be a power set, the CSP is classified as a set-CSP. Classical CSPs are such that the sets X and D cannot change. One useful abstraction on classical CSP (or set-CSP) is to view

[2] A graph property is hereditary if all its subgraphs respect it.

them as bipartite graphs subject to constraints [92]. Assigning a variable to a value can be represented by creating an edge between a variable vertex and and value vertex, as illustrated in Figure 5.2. The search space for these problems is hence the powerset of the set of edges in the complete bipartite graph. Configuration problems generalize the

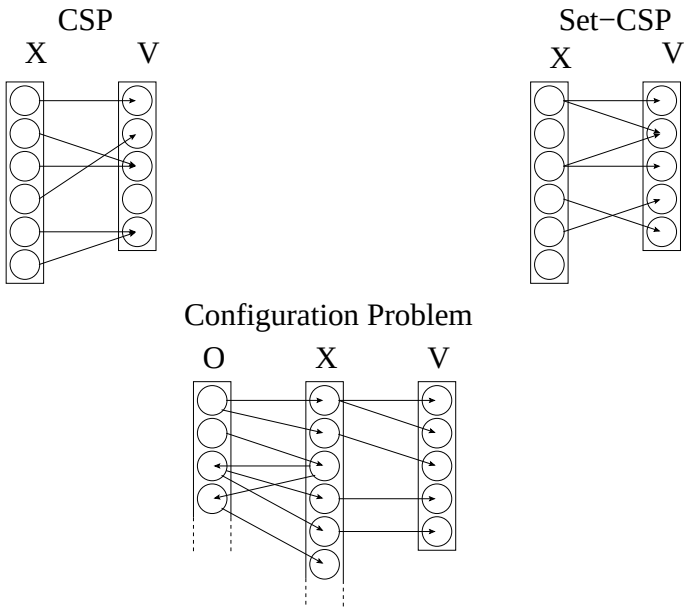


Figure 5.2: Graph abstractions for CSP, set-CSP and configuration problems

above. One pertinent abstraction is that of tripartite directed graphs, where the set of vertices partitions into a set O of objects, a set V of values and a set X of variables. Beyond the possibility to have attributes (a price, a size, etc.), objects can enter into various relationships with themselves or other objects. Edges from objects to variables denote either the object attributes, or the connections that the objects can establish towards other objects. Edges from variables to objects denote such connections. Edges from variables to values denote variable assignments, as in standard CSP. Unlike in the standard CSP case, the set of vertices in a solution is unknown, and potentially infinite. It should be noted however that once the decision set concerning the generation of the graph structure of a configuration solution is closed (hence the sets O and X are fixed), the remainder of the search amounts to a standard CSP problem. Of course for heuristic reasons, graph related assignments and attribute value assignments will generally be interleaved in configuration search.

CSP variants that address configuration

In many cases, a bound on the size of solutions of a configuration problem can be inferred from the cardinalities of the relations that exist among components. In such a situation, no infinite model exists, and fixing the numbers of participant objects to the bound

value keeps the search complete. We presented in Chapter 2 extensions to the CSP formalism that were proposed to allow to "add" variables during search within a fixed bound. For instance Conditional or Composite CSPs. The main drawback in using CSP or their variants is that most often, solutions contains fewer objects than their pre-defined number. Unused objects yield unwanted symmetries and filtering effort. Moreover, these extensions of the CSP formalism do not address the semi-decidable nature of the configuration problem, that can be best understood by noting that some configuration problems only have infinite solutions[3].

Symmetry breaking

Symmetry in CSP problems has been the focus of intense research effort in the past years, with significant achievements in efficient algorithms and in understanding the problem nature. Symmetry breaking methods fall into two broad categories: static and dynamic. First, to get rid of this, it can be observed that in some cases some static symmetry breaking can be performed by finding an appropriate model. For instance, using set variables can help collapse variable symmetries [4].
More generally, static symmetry breaking requires posting redundant constraints before the search begins. The effect of the constraints will be to select one *canonical* member for each equivalence class of partial assignments (including solutions and counter models of course). Such constraints can obviously be ad-hoc (eventually hand coded) or generic. [34] proves that static symmetry breaking for CSP with partial variable and partial value symmetries is possible using a number of constraints that remains linear in the problem size, although they remove a super exponential number of symmetries. The main drawback with static symmetry elimination techniques is that they may (and generally will) conflict with heuristic search. The main advantage of static symmetry breaking is that detecting that a current partial assignment is non canonical and should be rejected can be performed without any reference to the search history.
Dynamic symmetry breaking requires to detect and filter out symmetries during the search. Symmetry Breaking During Search [40] (SBDS) operates by adding constraints to the current node after each backtrack. The constraints prevent producing paths in the search tree that are isomorphic to a previously (eventually implicitly) generated one, according to the specification of an automorphism group. Dynamic lex constraints [85] also overcomes the potential conflict with heuristics of static symmetry breaking, and are shown to always explore fewer nodes than SBDS. In contrast to SBDS, Symmetry Breaking by Dominance Detection [30, 35] (SBDD) works by detecting at every choice point whether the current partial assignment is subsumed up to isomorphism by a previously explored node. This approach hence does not require to dynamically add constraints, and the number of dominance tests to perform is linear in the number of

[3]Example: imagine a representation of the sentence "Every man has a father" as a configuration problem, under the adequate semantics of the irreflexive transitive closure of the implicit "ancestor" relation.

[4]Example: when modeling a crew, one set variable can be used for the personnel instead of several symmetric separate variables.

variables. A generic SBDD framework based upon group theoretic results was presented in [39]. The authors acknowledge the fact that group theory computations at each node can prove extremely heavy in some cases. This is a motivation for the results presented in [92]: a polynomial algorithm for detecting dominance in standard CSPs with partial variable and partial value symmetries[5]. Very unfortunately, [92] proves that the problem of dominance detection is already NP-complete in the set-CSP case. In the more general configuration scheme, dominance detection amounts to the general subgraph isomorphism problem, also known as NP-complete.
Several approaches were experimented to tackle configuration isomorphisms, generally by reasoning at a single level. An obvious possibility is to prevent redundant connections of interchangeable objects[6] during search. Also useful is the replacement of objects by type counters [65], when the target instances remain un-distinguishable and are not themselves configured[7].
Meinolf Sellman and Pascal Van Hentenrick declare in [92] : "We believe that developing fast algorithms that (approximately) solve the dominance detection problem in NP-hard cases should be a focus of symmetry breaking research". From a complexity standpoint, dominance testing in configuration amounts to subgraph isomorphism, which is NP-complete. The approach presented in this paper is a form of static symmetry elimination for configuration problems, using a generic global constraint that prevents producing non canonical composition structures in pseudo linear time. Unlike the perspective in [92], we hence do not address a dominance detection problem. However, our work shares the objective of finding tractable approximations for symmetry breaking in the presence of set variables. We first isolate from a configuration problem the relations that participate to its *composition structure* (the term will be defined later, simply imagine a composite/component relationship without sharing, that yields tree-like structures). We also present an approximative algorithm for filtering out non canonical DAG expansion[8] of the canonical structure, which addresses the general configuration problem.

5.1.2 State graph of a configuration problem

We define the *state graph* $G_P = (X_P, E_P)$ of a configuration problem. The state set X_P contains all configurations (vertex-colored DAGs) corresponding to the structural model, and E_P are all the pairs (g, h) such that g, h $\in X_P$ and h is the result of a unit extension from g. (G_P is itself a DAG for which the root is the state (t, {1}, $\emptyset$, {(1,t)})). A structure generation procedure must be complete and non-redundant, i.e able to generate all structures of X_P only once while exploring a state graph G_P. The search itself can be represented with a covering tree T_P of G_P. Let us consider now the state graph G'_P, which is the subgraph of G_P containing only canonical structures. If

[5]The algorithm is based upon the Hopcroft Karp algorithm for computing perfect matchings in bipartite graphs.

[6]The not used yet instances of a same class ARE interchangeable.

[7]An example of a candidate situation is the modeling of a "purse": one does not require to create one "coin object" per coin. Only the number of coins of each type is meaningful.

[8]Note that DAGs can be used to model all configuration problems without loss of generality. It suffices to introduce in the model an extraneous "root" object, when this one is not initially present.

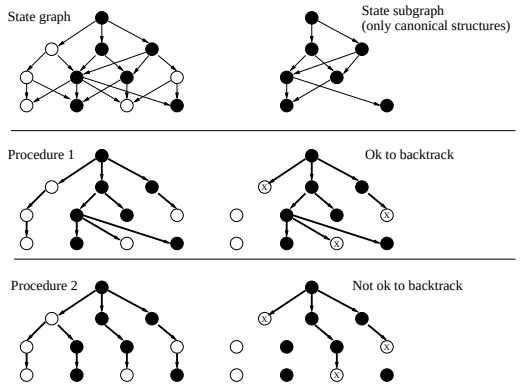


Figure 5.3: first line: a state graph G_P (left) et its sub graph G'_P limited to canonical configurations (right). Second line: case where the complete generation procedure yields a covering tree T_P (left) allowing to reach all canonical configurations although it backtracks on the non canonical (covering tree T'_P right). Third line: case when all canonical solutions cannot be reached.

valid, the canonical retractability property (remember that it means that each canonical graph has a canonical antecedent for unit extension) ensures that G'_P is connected and therefore the existence of at least one complete search procedure able to backtrack on non-canonical graphs. However, this does not imply that all search procedures will meet the requirements. Figure 5.3 illustrates the impact of the properties of the enumeration procedure on the possibility to backtrack at non canonical states. If the intersection T'_P between T_P and G'_P is not itself a connected graph, backtracking at non-canonical structures will yield an incomplete procedure. As a consequence, T'_P must be a covering tree of G_P. We will now present procedures respecting these criteria.

5.2 Isomorph-free tree structure generation

In this Section we present a generation procedure for canonical configurations that can be used when the structural model only contains composition relations. This procedure has been proposed in [44, 51]. We recall here their main results since our contribution is a direct generalization sharing the same formalisms. In particular, the procedure and context properties that we introduce in this Section will be useful in the generalization to graphs.
A *composition* relation between a type T_1 (called *composite*) and another type T_2 is a binary relation specifying that any object instance of T_2 can connect to at most one T_1 instance. As an example, the relation between `N` and `C` in Figure 5.1 is a composition relation, although this is not the case for the relation between `C` and `P`. In the composition case, solutions to the configuration problem can be represented with trees called *configuration trees.*

Definition 5.2.1 (T-tree) *A* T-tree *is a finite and non empty ordered tree where nodes*

are labeled by types and children are ordered according to the total order $\prec_{T_C}$. *The authors note* $(T, \langle c_1, \ldots c_k \rangle)$ *the T-tree with sub-trees* $c_1, \ldots c_k$ *and root label* T.

Proposition 5.2.1 *Let* A_1 *be a configuration tree,* C_1 *the corresponding T-tree, and* A_2 *the configuration tree rebuilt from* C_1. *Then* A_1 *and* A_2 *are isomorphic.*

5.2.1 A total order over T-trees

Configuration trees and T-trees being trees, they are isomorphic, equal, superposable, under the same assumptions as standard trees.

Definition 5.2.2 (Isomorphic T-trees) *Let* $C = (T, \langle a_1, \ldots, a_k \rangle)$ *and* $C' = (T', \langle b_1, \ldots, b_l \rangle)$ *be two T-trees.*
Isomorphism: *C and C′ are* isomorphic *(*$C \equiv C'$*) if* $T = T'$, $k = l$ *and there exists a bijection* $\sigma : \{a_1, \ldots, a_k\} \mapsto \{b_1, \ldots b_k\}$ *such that* $\forall\, i\ \sigma(a_i) \equiv a_i$.
Equality: *C and C′ are* equal *(*$C = C'$*) if* $k = l$, $T = T'$, *and* $\forall\, i\ a_i = b_i$.

Proposition 5.2.2 *Two configurations are isomorphic iff their corresponding T-trees are.*

As a means of isolating a canonical representative of each equivalence class of T-trees, the authors of [51] define a total order over T-trees. They define the following relations: $\preccurlyeq$ compares T-trees and $\preccurlyeq_{lex}$ is its lexicographic generalization to lists of T-trees.

Definition 5.2.3 (The relation $\preccurlyeq$) *Given two T-trees C = (T, L) and C' = (T', L'),* $\preccurlyeq$ *is recursively defined as follows: C* $\preccurlyeq$ *C' iff T* $<$ *T' or T = T' and L* $\preccurlyeq_{lex}$ *L'.*

Proposition 5.2.3 *The relations* $\preccurlyeq$ *and* $\preccurlyeq_{lex}$ *are total orders.*

Definition 5.2.4 (Canonicity of a T-tree) *A T-tree C = (T, L) is* canonical *iff L is empty or if L is sorted according to* $\preccurlyeq$ *and each c in L is itself canonical.*

Proposition 5.2.4 *A T-tree is the* $\preccurlyeq$*-minimal representative of its isomorphism class iff it is canonical.*

5.2.2 Enumerating T-trees

The rest of their study proposes on the one hand a procedure allowing for the explicit production of only the canonical T-trees, and on the other hand an algorithm to test and filter out non canonical T-trees. These two tools are meant to be integrated as components within general purpose configurators, so as to avoid the exploration of solutions built on the basis of redundant solutions of the inner structural problem of a given configuration problem. We continue in the sequel to call "configurations" the solutions of a structural problem. To generate a configuration tree amounts to incrementally build a T-tree which satisfies all structural constraints.

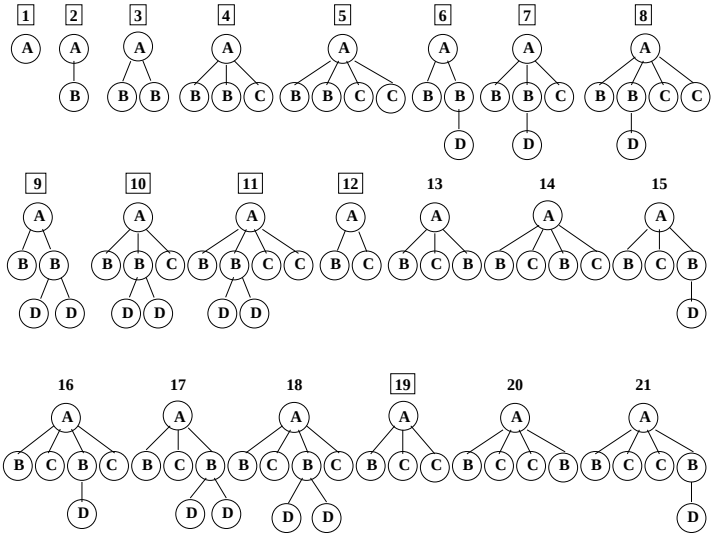


Figure 5.4: The first 21 T-trees ordered by $\preccurlyeq$, for a problem where at most two objects of type D can connect to an object of type B, two objects of types B may connect to an object of type A and two objects of type C may connect to an object of type A. The numbers of the $\preccurlyeq$-minimal representatives are framed.

Definition 5.2.5 (Extension, Unit Extension, Canonical Unit Extension) *An* extension *of a T-tree C is a T-tree C' which results from adding nodes to C. A* unit extension *is an extension which results from adding a single terminal node. If additionally both C and C' are canonical, the operation is called* canonical unit extension*.*

As explained before the search space of a (structural) configuration problem can be described by a state graph $G = (V, E)$ where the nodes in V correspond to valid (solution) T-trees and the edge $(C, C') \in E$ iff C' is a unit extension of C. The goal of a constructive search procedure is to find a path in G starting from the tree $(t, \langle\rangle)$ (recall that t is the type of the root object in the configuration) and reaching a T-tree which respects all the problem constraints (i.e. not only the constraints involved in the structural problem).

Definition 5.2.6 (Canonical removal of a terminal node) *The canonical removal of a terminal node from a T-tree C not reduced to a single node consists in removing its rightmost leaf.*

Canonical removal is technically useful for inductive proofs in the sequel.

Proposition 5.2.5 *Let G be the state graph of a configuration problem. Its sub-graph G_c corresponding to the only canonical T-trees is such that any canonical T-tree can be reached by a sequence of canonical unit extensions starting from the T-tree $(t, \langle\rangle)$.*

It immediately follows an important corollary:

Corollary 5.2.1 *There exist configuration generation procedures that filter out all the interpretations involving a non canonical structural configuration and remain complete.*

As shown in Figure 5.3, the Proposition 5.2.5 is a necessary but non sufficient condition for a T-tree generating procedure to be entitled the right to backtrack as early as it generates a non canonical T-tree. The Figure 5.5 assesses the crucial importance of this result: each canonical T-tree can be reached by adding a vertex and a node to a smaller *canonical* T-tree. For instance the canonical T-tree 11 is obtained by the canonical sequence $11 - 10 - 7 - 6 - 3 - 2 - 1$.

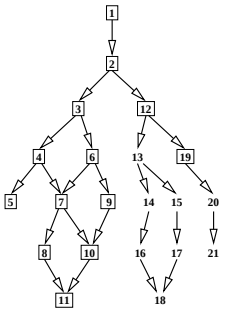


Figure 5.5: A portion of the state graph of unit extensions starting from the empty T-tree. Each canonical T-tree is represented by its number from Figure 5.4 . It is marked by a rectangular box if it is canonical.

Note that a canonical T-tree can generally be built from several distinct canonical T-trees. For instance, “11” can be obtained from either “8” or “10”. If the Proposition 5.2.5 is not verified, then testing T-tree canonicity cannot be used for backtracking since the only way to reach some canonical T-trees is by extending some other non canonical one. This a posteriori justifies their choice of canonicity, among the many possible such definitions for labelled trees. The same property will be guarantied in our generalization to graphs.

5.2.3 A canonicity testing algorithm and a generation procedure

The authors of [51] propose a procedure which tests the canonicity of a T-tree from the definition 5.2.4. They also propose a procedure **generate-tree** which generates all possible canonical structures, and prove that its time complexity is $O(n \log n)$. This procedure is complete, non-redundant and generates exclusively canonical structures.

5.3 Isomorph aware DAG generation

A configuration problem where only composition relations are involved can be filtered for isomorphisms by a constraint implementing the canonicity test. However, practical configuration problems also involve non composition relations. For instance, a relation

stipulating that m objects of a given type may connect to at most n objects of another type yields a bipartite graph structure[9]. Because edges can be added to a structure between preexisting nodes, non composition relations yield plain graphs as their models, which can be viewed as directed acyclic graphs (DAGs) if a root node is considered. Since any configuration problem can be adapted to involve such a root object, and without loss of generality, the full structural isomorphism problem for configurations *can be viewed as a DAG isomorphism problem.*

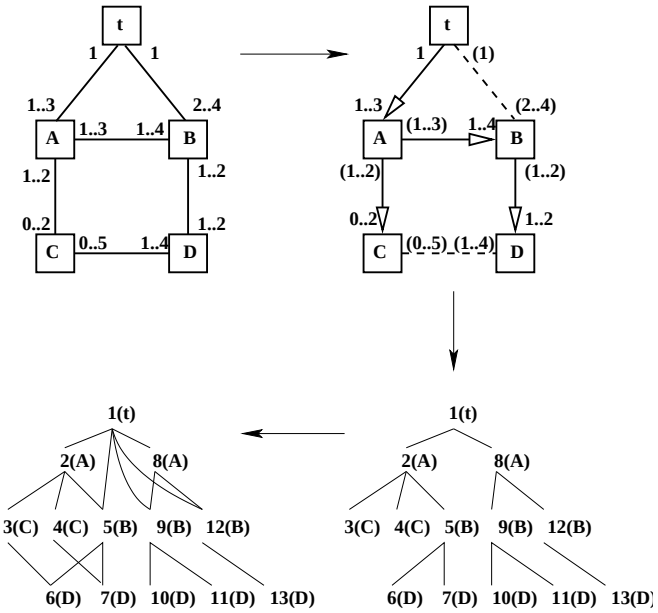


Figure 5.6: Generating DAGs from trees. In the upper left, a configuration model. Upper right, a composition covering tree of the model. Bottom right, a solution of the relaxed model. In the bottom left, a corresponding real solution after tree completion.

We now present an instance of a procedure generating only what we will call *weakly canonical DAGs*, defined as DAGs for which the minimal covering tree for the order $\preccurlyeq$ is canonical. As the permutation that would make its covering tree canonical is the same that would make the DAG weakly canonical, this avoids generating all non weakly canonical DAGs[10].
The leading idea is to first generate a canonical tree, called the *structure tree*, then perform unit extensions that solely create internal edges. As presented before, we can generate all canonical trees very efficiently. From such canonical trees, we generate all the DAGs sharing it as a structure tree, by adding internal edges.
The Figure 5.6 illustrates this idea. We start from a structural model containing general binary relations, from which we extract a sub-model having only composition relations[11].

[9]The bipartite graph isomorphism problem amounts to graph isomorphism
[10]The tractable generation of only one DAG per isomorphism class is an open problem.
[11]This sub-model is a covering tree of the original model!

The trees solution of this sub-model can be completed to produce solution DAGs of the original problem. Note however that the composition sub model is not pre computed, but implicitly defined during search by the choices made in generating the tree structure. We may thus generalize to DAGs several former notions:

Definition 5.3.1 (T-dag) *A T-dag is a finite directed acyclic graph with nodes labeled by types and neighbours ordered according to* $\prec_{T_C}$.

Testing the canonicity of a DAG amounts to the graph isomorphism problem. An algorithm like Nauty [71] is efficient in practise for this purpose, but its definition of canonicity does not match[12] the connexity requirement of Proposition 5.3.2. A backtrack search procedure for the enumeration of T-dags cannot hence remain complete if it fails when a non canonical structure in the sense of [71] is generated, which forbids using this definition[13]. Despite these difficulties, we now show how the canonicity of T-trees can be easily exploited to achieve at limited cost a weaker form of canonicity having useful properties, in the case of T-dags.

Definition 5.3.2 (Unit extension, extraneous, structural edge) *A* unit extension *of a T-dag is obtained by adding a single edge, either between two existing nodes - this is an* extraneous edge *-, or between a preexisting node and a new one - here called a* structural edge.

Definition 5.3.3 (structure T-tree) *The* structure T-tree *of a T-dag is the covering T-tree built from the sole structural edges having introduced a new node during its incremental constitution.*

Definition 5.3.4 (Weak canonicity of a T-dag) *A T-dag is* weakly canonical *if its structure T-tree is canonical.*

Proposition 5.3.1 *Every T-dag has a* weakly canonical isomorph.

Proof 5.3.1 *Let D be a non weakly canonical T-dag. There exists a permutation of its nodes yielding a canonical equivalent of its structure T-tree. This permutation hence maps D to an equivalent weakly canonical T-dag.*

It follows that a search procedure which only generates weakly canonical T-dags remains complete. Unlike with T-trees however, two weakly canonical T-dags can be isomorphic: such a procedure does not fully prevent from generating some isomorphic configurations.

Proposition 5.3.2 *Let G be the state graph of a configuration problem. Its sub-graph* G_c *obtained by removing all non weakly canonical T-dags is connected.*

[12] To the best of our knowledge, the existence of a canonicity definition for DAGs (or graphs) that would match this requirement is an open problem.

[13] Note that no accurate definition of canonicity is given for nauty which in that respect remains obsure (the reader is directed to the source code).

Proof 5.3.2 *It amounts to proving that any canonical T-dag can be reached by a sequence of canonical unit extensions from the starting T-dag (where only the root type occurs as a single node), or that (taken from the opposite side) the weak canonicity of any T-dag is preserved by removal of at least one of its edges. The operation of removing an extraneous edge obviously preserves canonicity since the structural T-tree remains unchanged. Furthermore, if a T-dag has no extraneous edge, it is a T-tree. The proposition hence holds because Proposition 5.2.4 ensures that the state graph of canonical T-trees is itself connected.*

We now present an instance of a procedure generating only weakly canonical T-dags. The idea is simple: first generate the structure T-tree, then perform unit extensions that solely create extraneous edges. We can generate all canonical T-trees using the generation procedure presented in [51]. From such canonical T-trees, we generate all the T-dags sharing it as a structure T-tree.
This procedure must however be implemented carefully to prevent from generating the same DAG multiple times. First, the possible extensions of a tree are ordered according with some order $<$. Edges are always added according with $<$ and an edge e cannot be added anymore if there exists an edge e' already added and $e < e'$. As for trees, it is obvious that this discards a certain amount of redundancies. Let $\mathbf{a}$ be the set of possible internal edges on a tree T, the number of DAGs that can be generated from T will be $2^{|\mathbf{a}|}$ instead of $|\ \mathbf{a}\ |^{|\mathbf{a}|}$. This however does not suffice to remove all redundant DAGs. To achieve this, and for each newly generated DAG, we search for the existence of a covering tree being ($\preccurlyeq$) less than the current structure tree, but not necessarily canonical. This situation naturally arises because after inserting new internal edges, the least covering tree may change. In that case it means that the current DAG can be discarded whether the found covering tree is canonical or not. Indeed there exists a canonical tree that is isomorphic to it, and thus the current DAG (or an isomorphic one) is already obtained by completion when this canonical tree is generated (and our tree generation procedure ensures that it has been or will be generated during the search).

Alternative structure tree search algorithm

At each newly created DAG (generated from tree T), we build the canonical covering tree T' by doing a depth-first search on the DAG (This test is called compare-mct($G \cup e_i$, T) in the following algorithm). If at one point, the selected edge differs from T, the DAG is rejected as it means the current working tree T is not the canonical one anymore. the time complexity of this procedure for finding is that of depth-first search in the worst case: O(n). To illustrate this, in the tree number 15 in Figure 5.8, the internal edge connecting the first `C` to the second `P` must not be inserted, since the smallest ($\preccurlyeq$) covering tree becomes the tree number 14.

Proposition 5.3.3 *Our procedure* `generate` *with a call to* `completion` *(see fig. 5.1) at each canonical tree generates only once each weakly canonical DAG.*

```
procedure completion(G, F, T)
   output G
   // Generate the set E = {e_1, ..., e_|E|}
   // of model-acceptable unit-extensions which are not not in F
   E = all-unit-extensions(G,F)
   for  i := 1 to | E | do
      if compare-mct(G ∪ e_i,T) then completion(G ∪ e_i, F ∪ {e_1,... , e_i}, T)
```

Table 5.1: The procedure **completion**.

Proof 5.3.3 *Firstly, the trees produced by* `generate` *are canonical and all different. Secondly, the procedure* `completion` *starts from a canonical T-tree C and adds edges in all possible ways, forbiding redundancy thanks to F. So, the procedure* `completion` *never generates the same T-dag twice.*
Thirdly, the procedure `completion` *rejects any T-dag which minimal covering T-tree is not C. T-dags that have not the same minimal covering T-tree are different.*
So, a T-dag cannot be generated twice by calls to the procedure `completion` *on two different T-trees.*

5.4 Exploiting symmetries

The procedure `completion(G)` can be further improved to eliminate some isomorphic DAGS resulting from unit extensions. The intuition is as follows: if the internal edges e_1 and e_2 that can complete G lead to two isomorphic graphs G1 and G2, then we forbid the unit extension e_2.
For example, adding the edge (4,3) to the DAG on the bottom right of Figure 5.1 produces a DAG isomorphic to the one obtained by adding edge (6,3). We might want to avoid one of the two extensions.
One expensive approach is to consider each pair of graphs completed with an edge from the set E of valid extensions, and test whether they are isomorphic or not (using Nauty for instance). In case they are, we delete from E one of these edges. The major drawback of this method is that there are potentially $O(n^2)$ unit extensions for a graph with n nodes, that is $O(n^2)$ that can be canonicaly labelled (thanks to Nauty for instance), thus leading to $O(n^4)$ pairs of canonical graphs to be compared (or $O(n^2 \log n)$ comparisons if we sort the graphs). In addition, even if Nauty has a polynomial behaviour on most graphs, it still has an exponential complexity in the worst case which disqualifies its use for large configuration problems. We henceforth use an incomplete method for removing such isomorphisms, by using the automorphism group (ie, the set of symmetries) of the current DAG: the covering trees of the DAGS are canonical, hence all their subtrees are $\preccurlyeq$ sorted. Henceforth, at any level in the tree, there may exist nodes equal wrt. $\preccurlyeq$. They are interchangeable, and are immediate neighbours, and all their sub-trees are pairwise interchangeable.
Although node interchangeability is costly to detect in the general case of unrestricted

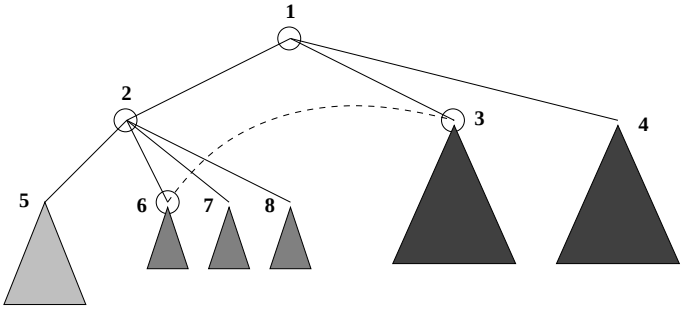


Figure 5.7: Adding an internal edge and marking

graphs, it is fast and obvious in the case of canonical trees. Testing whether two sub-trees having the same parent are interchangeable simply consists in testing if they are identical, an operation of time linear complexity. As a consequence, marking which node pairs are interchangeable in a tree is an operation in $O(n^3)$ that can be done at once before the completion of a structure tree.
To account for the fact that interchangeability is lost by nodes newly connected by an internal unit extension, we introduce a Boolean marker. The connected nodes must be marked, as well as the whole list of their parents up to the root of the tree. The marking is illustrated in Figure 5.7 by small circles around the nodes. A search procedure can reject all DAGs in which a newly inserted internal edge results in marking a node not being the leftmost in its equivalence class of interchangeability.
In the canonical tree represented by Figure 5.7, the trees rooted in nodes 6, 7 and 8 are identical, and so are the trees rooted in nodes 3 and 4. If the choice of interconnecting nodes from this two groups must be made, the search procedure can select only nodes within the trees 3 and 6. No node appearing within the sub-trees rooted in 4, 7 and 8 can be connected by a newly inserted internal edge. Once a connection between 3 and 6 is established for instance, node 3 loses its interchangeability with 4, and 6 loses its interchangeability with 7 and 8.
The figure 5.8 illustrates an instance of a state graph for the example in Figure 5.1. This problem requires generating a connection structure in the form of a bipartite graph[14]. This example shows that we can eliminate a significant number of redundant structures. Our experimental results show that the gains further increase quickly with the problem sizes. In this figure, only the structure T-trees are represented, but the extraneous edges that can be potentially added are drawn as dotted lines. The two framed structures are non canonical: the topmost hence yields a backtrack, and the "son" never gets generated. Within each T-dag, interchangeable nodes are shown using a circle around their class. For instance, in the last line, third tree from the left, the extraneous edge using the third "C" is forbidden, and the corresponding redundant DAG will never be generated. Here, exploiting isomorphisms and explicit automorphisms allows for generating only 27 among the 35 possible T-dags.

[14]Again, this problem is graph iso complete

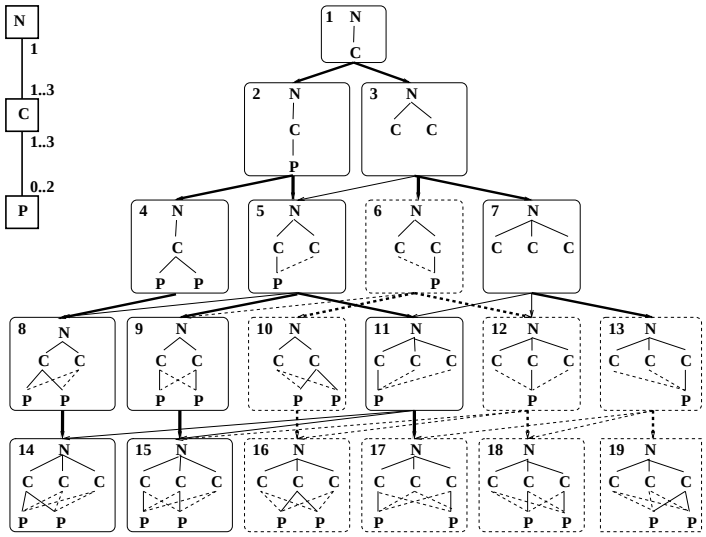


Figure 5.8: A portion of the state graph for the network configuration problem. Nodes are labeled with their type alone. Trees framed in dotted lines are not canonical. Dotted lines joining nodes inside frames denote possible complementing internal edges. All edges of the state graph denote unit extensions. Edges between non canonical trees are dotted. Bold edges are explored by procedure without canonicity check. Only continuous and bold edges are transitions explored by the procedure `generate`.

5.5 Experimental Results

Experiments on graphs

Other experiments were conducted for the computer-printer planning problem illustrated in Figure 5.1, on a 1.7 Ghz PC with 512M RAM, under Linux. In this example, solution structures are no longer restricted to trees. We have chosen this simple problem because it is generic: it involves a cardinality constrained relation between two types, which occurs very frequently in configuration problems. It must not be seen as a real application example, but rather as a way to reveal the interest and efficiency of such a procedure for eliminating isomorphisms. Indeed, the results on real problems involving many relations would benefit from the gain on each relation. For each choice of numbers of printers and computers, we have generated all DAGs using two algorithm variants: *Covering Tree* or *ct* (generation of canonical trees, each being completed to DAGs using an ordered set of possible extensions and backtrack on DAGs that have a covering T-tree less than the current) and *ct+treeInter* (*ct* + backtrack on (tree) equivalent internal edges for interchangeability). We compare the number of graphs generated by both algorithms with the number of graphs that are a solution of the problem. There are as many of them as the number of bipartite graphs (canonical or not) joining a set of c vertices to p vertices: $2^{c \cdot p}$.

C	P			*ct*			*ct+treeInter*		
		all graphs	structure trees	graphs	ratio	time	graphs	ratio	time
1	3	8	4	0		0	0		0
2	3	64	10	15	23.44%	0	8	12.5%	0
3	3	512	17	141	27.54%	0	67	13.09%	0
4	3	4096	24	1071	26.15%	0.05	446	10.89%	0.03
5	3	32768	31	8121	24.78%	0.13	2957	9.02%	0.12
6	3	262144	38	62931	24.06%	0.45	20920	7.98%	0.22
7	3	2.1 10^6	45	495117	23.57%	2.55	138719	6.61%	0.9
8	3	1.6 10^7	52	3927687	24.57%	31.99	965186	6.03%	6.21

Table 5.2: Results for the (C) PC - (P) printers problem. (times in seconds)

From Table 5.2 we see that the number of DAGs is significantly decreased when using the *ct* algorithm, due to the large number of avoided isomorphic DAGs. The *ct+treeInter* algorithm provides a good cut in the number of isomorphic DAGs, and overall computation time is also noticeably decreased.
Existing configurators are restricted to problems of limited size. Using these strategies lets us address larger problems, while avoiding the generation of useless solutions. Our computer/printer test problem should not be seen as artificial: any binary relation in an object model implies that a certain number of structures contain bipartite sub-graphs. The canonicity test for such graphs is graph iso complete, and current configurators would generate the graphs corresponding to the *all graphs* column of Table 5.2. These early results show that we can generate significantly fewer DAGs when the model involves only one binary relation. Should there be more than this (this is the common situation), the overall gain factor would benefit from individual gains, and in the particular case of a tree structural model it would be the product of the gains on each relation.

Using Nauty in conjunction with our graphs procedure

As stated previously, we only use a part of the interchangeable nodes as their detection in a general graph is a NP problem. We thus compute them in linear time on the trees before completion, and this interchangeability information is lost during completion when an edge is added to one of those nodes, or their childs. It is however interesting to observe the results when using Nauty in conjunction with our procedure by computing automorphism groups each time a graph is generated. We use again the computer-printer problem but defined in a slightly different way: instead of constraining the maximum number of printers, we constrain how many printers (m) can be connected to a computer, as shown in the object model 5.9. The results are presented in Table 5.3.
Those experimental results are contrasted. In the one hand, we observe that using nauty considerably increases the overall time, and quickly becomes unpracticable. On the other

n	m		*ct+treeInter*		*ct+nauty*	
		structure trees	graphs	time	graphs	time
2	3	14	11	0.002	10	0.036
2	4	20	23	0.017	20	0.076
2	5	27	42	0.023	35	0.14
2	6	35	69	0.024	56	0.24
2	7	44	106	0.042	84	0.387
2	8	54	154	0.050	120	0.623
2	9	65	215	0.06	165	0.906
3	3	34	200	0.027	121	0.295
3	4	55	872	0.06	393	1.052
3	5	83	3329	0.08	1073	2.994
3	6	119	11588	0.249	2594	7.623
3	7	164	37796	0.481	5689	18.01
3	8	219	116506	1.167	11557	40.46
3	9	285	344637	3.4	22026	91.748
4	3	69	4547	0.161	1970	0.573
4	4	125	65021	0.555	11968	24.147
4	5	209	821392	5.668	73136	152.045

Table 5.3: Results for the (n) PC - connected to (m) printers problem. (times in seconds)

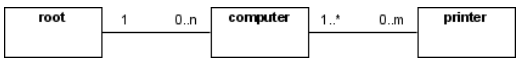


Figure 5.9: Object model for a computer-printer problem.

hand, we can see the *ct+treeInter* is not able to reject a large number of isomorphism, but the time spent grows in a slower curve. Using Nauty in conjunction with our procedure remains a reliable possibility: the canonical trees generation together with the minimal covering tree method allows to quickly reject a large number of graphs, leaving much less graphs for the costly nauty automorphisms computation. It is obviously more efficient than generating all possible graphs and pruning the results with isomorphisms tests. Finally, the large number of isomorphisms not detected by the *ct+treeInter* procedure indicates that other algorithms may exist to maintain a greater part of interchangeability information in a linear time, considering that only one node or edge is added at each step.

Insertion in a general configuration search

A configuration problem statement normally involves classes, relations, and constrained attributes. Generating the configuration structure is hence a fragment of the whole problem. Our approach is interesting in several respects in this general case. On the one hand, once a structure has been generated, the problem amounts to a static configuration problem, hence amenable to usual techniques including CSPs. Also, as shown before, the automorphism group of the built structure is easily exploited. Further search may benefit from this in the process of instantiating attributes or classifying objects.

5.6 Conclusion

This research is a generalization to DAGs of existing work on trees. It greatly extends the possibilities of dealing with configuration isomorphisms, until today limited either to the detection of the interchangeability of all yet unused individuals of each type or to the use of non configurable object counters.
The generation procedure for vertex colored DAG structures that we have presented addresses the structural isomorphism problem of configurations and allows for important gains for any configuration problem, even of small size. Not all the non canonical structures are discarded, however, the algorithms used are time pseudo linear, which allow for their seamless integration within an enumerative search procedure. Indeed, the weak-canonicity test that we presented can be inserted as a redundant constraint in generic configuration solvers, which allows to extend the range of practically tractable problems. This is a topic of future work.
Once the canonical configuration structure is known, the interchangeability of a number of nodes can be readily exploited by the remaining search, which involves decisions

relative to object classification and attribute variables. The corresponding (partial) automorphisms are known as a side effect of canonicity testing, and thus induce no overhead.
Since there are are still a large number of isomorphisms left undetected, another perspective is to discover other time pseudo linear tests able to reject them.

Chapter 6

Stochastic Search for Configuration: Ant Colonies

An inherent difficulty in enumerative search algorithms is the combinatorial explosion that occurs when increasing the size of the input. This is especially true for optimization problems, where it is not only required to find a solution but also to find the best one according to some evaluation function. A large field of research dedicates to methods which limit this explosion in exhaustive search algorithms: filtering, decomposition, heuristics, etc. We have seen the benefits of symmetry-breaking in constraint programming and tried to apply it to configuration in the previous chapter.
Another field of research uses *incomplete algorithms* which only partially explore the search space. In particular, *stochastic* algorithms use a combination of random and heuristic methods in order to quickly find a solution. The counterpart is that the inability to find a solution does not prove that none exists. Among other stochastic methods, Marco Dorigo [17] proposed a meta-heuristic for combinatorial optimization problems inspired from the behaviour of biological *ant colonies* known as *Ant Colony Optimization* (ACO). It has later been applied to CSPs problems [98, 100] with competitive results on a wide range of problems.
The aim of this chapter is to study the usability of an ACO-based algorithm for configuration problems. It is, to the best of our knowledge, the first attempt on using stochastic methods in configuration. We begin with a presentation of the ACO algorithm mechanism and its application to CSPs. We then describe how the specificities of configuration impacts the ACO approach, in particular with regards to its dynamic nature. Based upon these observations we propose an ACO framework for configuration with two algorithms variants. Finally we describe a java implementation and provide preliminary experimental results.

6.1 Introduction to ACO

Research on ants behaviour has shown that their communications are mostly based on the use of a chemical agent they produce called *pheromones*. A particular type of

pheromones is the *trail pheromone* deposited on the ground. The marking of a path, for example from a food source to the nest, is then followed by other ants with some random or local visibility fluctuations. This type of indirect communication via the environment is known as *stygmergy*. Furthermore, pheromones are volatile, meaning that laid trails evaporate over time which allows for *diversification* and *path exploration*.
Biological experiments have shown that ants can make use of trail pheromones to use the shortest path to a food source. When two paths of different length are offered to a colony, the ants returning from the shortest one deposit pheromones earlier leading to a majority of ants choosing this path and thus increasing its pheromones concentration. This is known as a *distributed autocatalytic process*.
The shortest path problem shares similarities with many computational problems such as the well-known Traveler Sales Problem (TSP), which led to inspire from the ants behaviour to design new solving algorithms [27]. Later researches extended the approach to a *meta-heuristic* for discrete optimization problems [26]. A recent and comprehensive study of ACO can be found in [25].

6.1.1 ACO meta-heuristic and algorithms

ACO and the Ant System

In [26], Marco Dorigo proposes an ACO meta-heuristic and an algorithm, known as the Ant System, for combinatorial discrete optimization problems. We present in the following his definition of the problem and his description of the meta-heuristic.

Definition 6.1.1 *A combinatorial discrete optimization problem*
A model $P = (S, \Omega, f)$ of a combinatorial discrete optimization problem consists of:

- *a search space S defined over a finite set of discrete decision variables $X_i, i = 1, ..., n$.*
- *a set of constraints Ω among the variables.*
- *an objective function $f : S \rightarrow \mathbb{R}_0^+$ to be minimized.*

The generic variable X_i takes values in $Di = \{v_i^1, ..., v_i^{|Di|}\}$. A feasible solution $s \in S$ is a complete assignment of values to variables that satisfies all constraints in Ω. A solution $s^* \in S$ is called a global optimum if and only if: $f(s^*) \leq f(s) \, \forall \, s \in S$.

The problem model is associated in ACO with a pheromone model: a pheromone value is associated with each possible assignment of a value to a variable. Formally, the pheromone value τ_{ij} is associated with the solution component c_{ij} , which consists in the assignment $X_i = v_i^j$. The set of all possible solution components is denoted by C.
A *construction graph* $G_C(V, E)$ can be obtained from C where the vertices V or the edges E are the solution components. An artificial ant builds a solution by traversing the fully connected construction graph from vertex to vertex. Furthermore, ants deposit

pheromones on the components by a certain amount $\Delta\tau$ depending on the solution quality.
An algorithm for the ACO meta-heuristic is given in Table 6.1.

Termination conditions: the usual conditions are either that a solution is found (in

initialize parameters and pheromone model
while termination conditions not met do
ConstructAntSolutions
UpdatePheromones
endwhile

Table 6.1: The ACO meta-heuristic.

the case of satisfiability), or a given quality is achieved(in the case of optimization), or a given number of iterations has been done. We also find combinations for instance a number of iterations without improvement of the solution.
ConstructAntSolutions: a set of m artificial ants constructs solutions starting from an empty partial solution $s^p = \varnothing$. At each construction step, it is extended by adding a solution component c_{ij} from the set $N(s^p) \subseteq C$ where $N(s^p)$ is the set of components which can be added without violating the constraints in Ω. The choice of a component is a probabilistic choice influenced by pheromones and heuristics (corresponding to the biological local visibility of an ant). It can vary in different versions of ACO algorithms. In *Ant System*, the first proposed algorithm for ACO, the probability to select a component c_{ij} for an ant k is:

$$p_{ij}^k = \frac{\tau_{ij}^{\alpha}.\eta_{ij}^{\beta}}{\sum_{c_{il} \in N(s^p)} \tau_{il}^{\alpha}.\eta_{il}^{\beta}}$$

where the parameters α and β control the relative importance of the pheromone versus the heuristic information η_{ij}.
UpdatePheromones: update the pheromones values so that good solutions are favored in the following iterations. This step usually consists in applying a pheromone evaporation to all values and then increasing the values associated with good solutions. In Ant System, the pheromone value τ_{ij} is updated as follows:

$$\tau_{ij} \longleftarrow (1-\rho).\tau_{ij} + \sum_{k=1}^{m} \Delta\tau_{ij}^k$$

where ρ is the evaporation rate and $\Delta\tau_{ij}^k$ is the quantity of pheromone laid on c_{ij} by ant k:

$$\Delta\tau_{ij}^k = \begin{cases} Q/f(s_k) & \text{if ant } k \text{ used component } c_{ij} \text{ in its solution } s_k \\ O & \text{otherwise} \end{cases}$$

where Q is a constant and f is the objective function.

Parameters

The effectiveness of ACO-based algorithms is to a large extent dependent on the values of different parameters. Optimal values depend upon the application domain but a few

general patterns have emerged. Experiments have shown that decreasing the value of α or ρ emphasizes exploration, which leads to better solutions but increases running time. Concerning the number of artificial ants at each iteration, a reduced number decreases the quality of solutions whereas an upper bound can be found above which the quality of the solutions is not significantly affected.

Algorithm variants

There exists a large literature on variants of the Ant System. One of the best known improvements is the *MAX-MIN Ant System* (MMAS) [105]. In MMAS, only the best ant updates the pheromone values at each iteration with the following formula:

$$\tau_{ij} \longleftarrow \left[(1-\rho).\tau_{ij}^{best} + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{max}}$$

where the operator $[x]_b^a$ is defined as:

$$[x]_b^a = \begin{cases} a & \text{if } x > a \\ b & \text{if } x < b \\ x & \text{otherwise} \end{cases}$$

,τ_{min} and τ_{max} are respectively a lower and an upper bound imposed on the pheromones, and $\Delta\tau_{ij}^{best}$ is:

$$\Delta\tau_{ij}^{best} = \begin{cases} 1/f(s_{best}) & \text{if } c_{ij} \text{ is in the best solution } s_{best} \\ O & \text{otherwise} \end{cases}$$

where s_{best} is the best ant's solution. s_{best} can either refer to the *iteration best* (i.e the best solution at the current iteration), the *best-so-far* or a combination of both.
Some algorithms such as ACS [27, 38] apply a *local pheromon update* in order to diversify the search by subsequent ants in the current iteration. Others vary the value of parameters α, β and ρ over time so as to begin with a high exploration behaviour and later concentrate on refinement.
Hybrid algorithms which combine different methods have also been investigated. A *local search* can be applied to the solutions found by ants to improve their quality before the pheromones update. *Supervision* tasks can also be handled at the end of each iteration where actions that could not be carried by a single ant take place (for instance analyzing the current best solution versus the previous ones). ACO has been successfully applied to many application domains: vehicle routing, sequential ordering, scheduling, protein folding, bin packing, etc. The application to CSPs is of particular relevance as we have seen that configuration and CSPs share many similarities.

6.1.2 Application to CSPs

In [99], ACO is used for solving CSPs (we use the definition of CSPs given in Chapter 2). In the proposed approach, the vertices of the construction graph are the variable-value pairs $< X_i, v >$, and there is an edge between any vertices of a different variable.

The model associates a pheromone $\tau(< X_i, v >, < X_k, v >)$ to an edge. The algorithm follows the ACO meta-heuristic and includes $MAX - MIN$ improvements: bounds are defined on pheromones values and only the best ants update the pheromones. The construction of a CSP assignment, as defined in [99], is presented in Table 6.2.
Unlike the original ACO, the probabilistic choice does not depend solely on the previ-

procedure **construct assignment A for CSP (X,D,C)**
  $A \leftarrow \emptyset$
  while $\mid A \mid < \mid X \mid$ **do**
    Select a variable $X_j \in X$ that is not assigned in A
    Choose a value $v \in D(X_j)$ with probability $P_a(< X_j, v >)$
    $A \leftarrow A \cup < X_j, v >$

Table 6.2: The construction of an assignment in the CSP application of ACO

ously chosen variable, i.e here all visited vertices A are equally important:

$$\tau_A(< X_j, v >) = \sum_{<X_k, m> \in A} \tau(< X_k, m >, < X_j, v >)$$

The heuristic for selecting a value is inversely proportional to the number of new violated constraints. The heuristic for selecting the next variable to be assigned is the smallest-domain ordering (often called "first-fail" heuristic).
The approach presents competing experimental results, especially when it is associated with local search to improve ants solutions. [99] also describes a pre-processing step based on local search that can be used to initialize the pheromone trails.

6.1.3 Original properties of configuration

Configuration algorithms have original features which impact on potential ACO algorithms. We consider the choices that will be made by a configurator, using the definition of a configuration problem defined in Chapter 2, and discuss the related issues for artificial ants.

Classification

When a component is selected (or created by necessity), a configurator needs to classify it with respect to the model's taxonomy. Inheritance can be modelled so that all possible subtypes are leafs of the inheritance tree. The choice of a type then resolves to a discrete choice among a finite set of distinct elements. This decision has the same nature as the choice of a path in the original ACO, or the choice of a value for a variable in CSP.

Attributes

Attributes in configuration have a finite domain. Possible values of an object's attribute therefore resolve to a discrete value choice among a finite set of elements.

Dynamic structures

Since the structure of a configuration is not known from start, a configurator building a solution can dynamically create new components. The creation of components, in most configurators, is tied to the selection of a relation's target(s).

Relations cardinalities and targets

When a configurator builds an instance, it has to instantiate the participating object relations. This means to select the cardinality and target objects for each relation. In the general case where creation of objects upon necessity is allowed, the set of available targets for a relation is potentially infinite.
Selecting (or creating) the objects as a relation's target(s) is speccific in that it requires to select *a set* from a (possibly infinite) set of targets.
On the one hand, existing ACO's probabilistic choices must thus be adapted to deal with potentially unbounded sets of values (here, of targets).
On the other hand, we need to consider the selection of a set, where the selected set's size is the cardinality of the relation. We can imagine an iterative process where a configurator each time chooses between adding a new target or stopping the process. However such an algorithm may not terminate. Another approach is to first select a cardinality and then iteratively select the corresponding number of targets.
We thus eventually consider the selection of a cardinality. A relation's cardinality always has a lower bound (at least 0). If it also has an upper bound, then selecting the cardinality amounts to a discrete choice among a finite set of elements. If there is no upper bound then again we have the choice of a value from an unbounded set (this time, of integers).

Construction of a configuration with artificial ants

Based upon the described features, we first give an overview of how ACO can be adapted for configuration. Since solutions may have an unbounded number of components, problems do not comply with Definition 6.1.1. In particular, it is not possible to exhibit beforehand a construction graph.
The structure of a configuration instance can itself be seen as a graph. In this structure graph, the vertices are the participating objects and the edges are the relations between them. At a given moment of a configuration enumerative algorithm, the superimposition of all created structure graphs is also a graph. Pheromones can be laid on its edges, representing for an artificial ant the choice of a relation's target.
An artificial ant, in a given vertex of this graph, not only has the choice of following one or more existing edges, but may also create edges to new components. The number of edges (i.e targets of a relation), followed or created by an ant, is defined by the (unbounded) choice of a cardinality.
Finally, as each vertex is a component of the configuration instance, an artificial ant will have to classify the component and select a value for each of its attributes. These

classes and values can be seen as vertices only connected to the corresponding component's vertex.
The construction graph of ACO for configuration, made of all these elements, is therefore created dynamically during solving by superimposition of all created instances. Artificial ants move along (and extend) this graph to create solutions.

6.2 ACO for configuration

6.2.1 Pheromones model

Dealing with unbounded sets

Given the properties of configuration problems presented in the last section, in particular the presence of unbounded sets, we introduce a new pattern for an artificial ants probabilistic choice. The pattern allows to simulate the choice of a value from an unbounded set through an *evolving* finite set. The set evolves (i.e is increased for the next ACO iteration) when an artificial ant selects a value above a chosen *separator*.
We give an example with a relation r, $minCard(r) = 2$. We consider a starting set $R_{card} = \{2, 3, 4, 5\}$ with value 3 being a separator for the evolution of the set. If the chosen value is less than 3, the set remains unchanged. If the value 4 is chosen, the set is modified for the next ACO iteration into $R_{card} = \{2, 3, 4, 5, 6, 7\}$ where value 5 is the new separator.
We first give a formal definition of the general pattern then we show its application to the instantiation of a relation.

simu-finite sets : (stands for SIMUlated FINITE sets for unbounded SETS)
(1) We consider the selection of a value v from the finite set $V = \{v_0, \ldots, v_i, v_{i+1}, \ldots, v_j\}$ where:
$\{v_0, \ldots, v_i\}$ and $\{v_{i+1}, \ldots, v_j\}$ are finite sets of distinct possible values such that:
For all k, $i < k < j + 1$, we can construct a set $V^+ = \{v_{j+1}, \ldots, v_{j+k-i}\}$ of possible values such that $V^+ \cap V = \varnothing$.
(2) When a value v_k, $i < k < j + 1$ is selected, the set is modified into $V = V \cup V^+ = \{v_0, \ldots, v_k, v_{k+1}, \ldots, v_{j+k-i}\}$ for the next ACO iteration.

Instantiating a relation

We have seen that a possible approach for an artificial ant is to first select a cardinality r_{card} for a given relation r, and then iteratively select the corresponding number of targets. Using the simu-finite choice pattern, we thus introduce a probabilistic choice for unbounded cardinalities and another one for selecting a target when it is allowed to dynamically create such a target.

unbounded cardinality choice : An unbounded cardinality choice is a choice from an unbounded set of integers. We apply the pattern as follows:

(1) We consider the selection (for a relation r) of a value r_{card} from the finite set $R_{card} = \{r_{card_{min}}, \ldots, r_{card_{min+i}}, r_{card_{min+i+1}}, \ldots, r_{card_{min+i+d}}\}$ where:
$r_{card_{min}}$ is the lower bound of the relation's cardinality,
$\{r_{card_{min}}, \ldots, r_{card_{min+i+d}}\}$ is a continuous set of integers, i.e $r_k = r_{k-1} + 1$
$i >= 0$ is a chosen value at the beginning,
$d >= 1$ is a parameter controlling the size of the allowed increase for cardinality at each iteration.
(2) When a cardinality $card_j$ such that $min+i < j < min+i+d+1$ is selected, the set is modified into $R_{card} = \{r_{card_{min}}, \ldots, r_{card_{min+j}}, r_{card_{min+j+1}}, \ldots, r_{card_{min+j+d}}\}$ where the set of values $R^+_{card} = \{r_{card_{min+j+1}}, \ldots, r_{card_{min+j+d}}\}$ is constructed using a straightforward addition sequel, i.e $r_k = r_{k-1} + 1$.
We recall the example of a relation r, $minCard(r) = 2$. We start with $d = 3$, $i = 0$. We thus have the set $R_{card} = \{2, 3, 4, 5\}$. If the value $v_k = 4$ is chosen, the set is modified into $R_{card} = \{2, 3, 4, 5, 6, 7\}$ where $v_i = 5$ is the separator for the next ACO iteration.

target choice : The unbounded set of a target choices stems from the possibility to dynamically create an object. We apply the simu-finite pattern as follows:
(1) We consider the selection (for a relation r) of a value r_{t_i} from the finite set $R_t = \{r_{t_0}, \ldots, r_{t_i}, r_{t_{i+1}}\}$ where:
$\{r_{t_o}, \ldots, r_{t_i}\}$ is the set of *existing* objects which can be target of the relation
$r_{t_{i+1}}$ is the choice of creating a new object
(2) When the target $r_{t_{j+1}}$ is selected, the set is modified into $R_t = \{r_{t_0}, \ldots, r_{t_{i+1}}, r_{t_n}\}$ where $r_{t_{i+1}}$ is now an existing object, and r_{t_n} is the choice of creating a new object. Alternately, the algorithm's designer may decide to only allow the creation of one (or a limited number of) objects during the iteration. In this case the modified set does not include the value r_{t_n}.

We can now give the following definition of pheromones-based instantiation of a relation.

Definition 6.2.1 *Instanciation of a relation*

- *Instanciating a relation r means to probabilistically select a cardinality r_{card_j} from the simu-finite set R_{card}, and then iteratively select r_{card_j} targets from the simu-finite set R_t.*

- *the probability to select a cardinality r_{card_j} is:*

$$p_{r_{card_j}} = \frac{\tau^{\alpha}_{r_{card_j}} . \eta^{\beta}_{r_{card_j}}}{\sum_{l=0}^{min+i+d} \tau^{\alpha}_{r_{card_l}} . \eta^{\beta}_{r_{card_l}}}$$

where $\tau_{r_{card_j}}$ *is the pheromone value for the cardinality* $card_j$ *and* $\eta_{r_{card_j}}$ *the heuristic information.*

- *the probability to select a target* r_{t_j} *is:*

$$p_{r_{t_j}} = \frac{\tau^{\alpha}_{r_{t_j}} . \eta^{\beta}_{r_{t_j}}}{\sum_{k=0}^{i+1} \tau^{\alpha}_{r_{t_k}} . \eta^{\beta}_{r_{t_k}}}$$

where $\tau_{r_{t_j}}$ *is the pheromone value for the target* r_{t_j} *and* $\eta_{r_{t_j}}$ *the heuristic information.*

Classifying

We follow with the other types of choices done by artificial ants in a configuration problem. As classification is a choice from a finite set of elements, it is possible to use the classical pattern from ACO.

Definition 6.2.2 *Classification*

- *Classifying an object of type* t_i *means to probabilistically select one subtype from the set* $finalsubtypes(t_i) = \{t_i^j, \ldots, t_i^k\}$

- *the probability to select a type* t_i^j *is:*

$$p_{t_i^j} = \frac{\tau^{\alpha}_{t_i^j} . \eta^{\beta}_{t_i^j}}{\sum_{l=0}^{k} \tau^{\alpha}_{t_i^l} . \eta^{\beta}_{t_i^l}}$$

where $\tau_{t_i^j}$ *is the pheromone value for the type* t_i^j *and* $\eta_{t_i^j}$ *the heuristic information.*

Instanciating attributes

Again, we have a classic choice from a finite set of elements.

Definition 6.2.3 *Instanciation of an attribute*

- *Instanciating a variable* X_i *means to probabilistically select one value from the set* $D_i = \{x_i^0, \ldots, x_i^k\}$

- *the probability to select a value* x_i^j *is:*

$$p_{x_i^j} = \frac{\tau^{\alpha}_{x_i^j} . \eta^{\beta}_{x_i^j}}{\sum_{l=0}^{k} \tau^{\alpha}_{x_i^l} . \eta^{\beta}_{x_i^l}}$$

where $\tau_{x_i^j}$ *is the pheromone value for the value* x_i^j *and* $\eta_{x_i^j}$ *the heuristic information.*

Pheromones update

The update of pheromones values, similar to the one in $MAX - MIN$ Ant System, is based on the instance created by the best ant at each iteration:

$$\tau_{ij} \longleftarrow \left[(1-\rho).\tau_{ij}^{best} + \Delta\tau_{ij}^{best}\right]_{\tau_{min}}^{\tau_{max}}$$

where $\Delta\tau_{ij}^{best}$ is a value depending on the instance's quality. In the case of satisfiability, quality can be defined by the rate $\frac{numberOfConstraintsFulfilled}{totalNumberOfConstraints}$.
However in configuration original issues arise. Firstly, evaporation's main goal is to forget *bad* assignments but if an object o_i does not participate to the instance, it is not obvious if we will benefit from evaporating its associated decisions. We therefore propose an alternative to *total evaporation* with a *restricted evaporation*.
In a restricted evaporation, all pheromones associated to o_i (classification, attributes instantiation or relations having o as source) are left unchanged. Note that pheromones associated with a relation $o_j \rightarrow o_i$ are still evaporated thus considering the fact that the object has not been selected for the current best instance. We leave both evaporation alternatives as a parameter of our ACO implementation.
Secondly, we need to consider the dynamic pheromones of simu-finite sets. In the case of relations targets choices, the pheromones associated with the creation of an object are only updated if the best solution has effectively created a target during this iteration. The corresponding new element of the set is given the same pheromone value. In the case of the cardinality choice, dynamically added cardinality values have a shared pheromone initialization value.
Finally, different types of choices may require a different treatment. Therefore each parameter relative to pheromones is duplicated for each type of associated choice. For instance, the minimum allowed value can be different for classification or relations targets.

6.2.2 Algorithms

We propose two algorithms ($ACOC_{class}$ and $ACOC_{graph}$) to implement a configuration solver based on the presented pheromones model. Both of them follow the original ACO meta-heuristic algorithm but differ in the way an artificial ant constructs a solution. The shared part is given in Table 6.3.

$\mathbf{ACOC_{class}}$

$ACOC_{class}$ first selects a set of objects of each top-class (i.e without parent) that may participate in the solution. To achieve this, we create an artificial root object with relations to each top-class. These relations have a lower bound equal to 0, and no upper bound (unless the number of objects of this class is limited by the model). The previously defined instantiation of a relation is applied to this root object to decide which objects participate in the solution.

```
function instance configure
    for i := 0 to numberOfIterations do
        for j := 0 to numberOfAnts do
            instance:=generateInstance
            quality:=evaluateInstance
            if (quality>bestQuality) do
                bestInstance:=instance
                bestQuality:=quality
        updatePheromones
        updateSimuFiniteSets
    return instance
```

Table 6.3: The function **configure**

Then, for each object, $ACOC_{class}$ classifies it, instanciantes its attributes and finally instantiates its relations. When instantiating relations, an upper bound is set to the number of objects in the target class. Furthermore, the creation of objects is not allowed anymore during the selection of targets. The algorithm is presented in Table 6.4. Figure 6.1 gives a graphical example of its behaviour where a number represents a classified object with instantiated attributes.

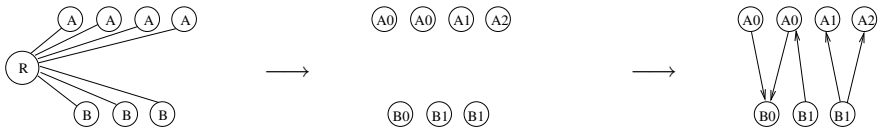


Figure 6.1: The 3 steps in $ACOC_{class}$: (1) select objects, (2) classify and instantiate attributes, (3) instantiate relations

$\mathbf{ACOC_{graph}}$

$ACOC_{graph}$ does not select beforehand the participating objects. The algorithm starts by classifying a chosen root node. It instantiates its attributes and relations and then classifies the selected targets. Each target object is then treated recursively following a depth-first approach. The algorithm is presented in Table 6.5. Figure 6.2 gives a graphical example where a number represents a classified object and "+" represents an object with instantiated attributes.

```
function instance generateInstance
    instance=initializeInstance
    for i := 0 to numberOfClasses do
        if (!classHasParent) do
            instantiateClassPool
    for i := 0 to numberOfObjects do
        classifyObject
        for j := 0 to numberOfAttributes do
            instantiateAttribute
        for j := 0 to numberOfRelations do
            instantiateRelation
    return instance
```

Table 6.4: The function **generateInstance for** $ACOC_{class}$

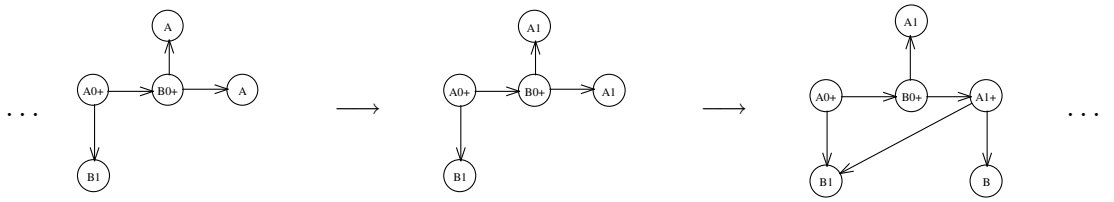


Figure 6.2: $ACOC_{graph}$: instantiate attributes, instantiate relations, classify new targets, iterate with a depth-first approach

Construction path

As explained before, our construction graph is not computed beforehand but defined by the superimposition of all instances generated by previous ants. Another significant difference with the original ACO meta-heuristic is in the *construction path* followed by ants inside this graph:

- In $ACOC_{graph}$, the construction path is related to the depth-first search in the structural part of the construction graph (i.e components and relations). However, at each vertex, an artificial ant will follow (or create) an edge, then return to the original vertex until all decisions for this component have been taken and all its targets have been configured. This behaviour is illustrated in Figure 6.2. Once the bottom objects "B" have been configured, the ant will return to the middle object "B0+" and move to the top object "A1".

- In $ACOC_{class}$, there is no explicit construction path. An artificial ant "jumps" from one component to another (if we leave apart the artificially created root node). As a result, the algorithm does not guaranty that the created structure

```
function instance generateInstance
    instance=initializeInstance
    constructionStack:=root
    classifyRoot
    while (constructionStack≠ ∅) do
        constructObject
        removeObjectFromConstructionStack
    return instance
procedure constructObject
    for i := 0 to numberOfAttributes do
        instantiateAttribute
    for i := 0 to numberOfRelations do
        instantiateRelation
        classifyTargets
        addTargetsToConstructionStack
    return
```

Table 6.5: The function **generateInstance for** $ACOC_{graph}$

graph is connected. Since this is often considered an implicit constraint in configuration, it needs to be forced either with an explicit model constraint, or during the solving by discarding unconnected instances.

Finally, in both algorithms, artificial ants choices are independent from previous decisions. In this respect, our approach is similar to the application to CSPs [99].

6.3 Implementation and Experiments

6.3.1 Implementation

The program used in our experiments has been developed using the Java language. It is composed of:

- a constrained object models library used to define the configuration problems. The object model library supports classes inheritance and unbounded relations. The constraints library supports universally quantified constraints as well as basic operators (numerical operators, classes operators),
- an instance library used to define instances of a COM,
- a pheromones model library used by artificial ants to deposit and follow pheromones,

- an implementation of the $ACOC$ algorithms.

Our library is independent from the solving engine and could therefore be used as a comparative platform between different finite model search algorithms for constrained object models. An example of how to use the library in the context of ACOC is given in Annex III.

6.3.2 Heuristics

In our experiments, we did not use any heuristic for value choices. Artificial ants are thus completely driven by pheromones.
Concerning variable choice heuristic, $ACOC_{class}$ uses a predefined (random) order whereas $ACOC_{graph}$ ordering is implicit by its depth-first nature. In both cases classification and attributes are treated in a predefined random order, as well as different relations from the same object.

6.3.3 Parameters and Particle Swarm Optimization

Parameters have a large influence on the behaviour of ACO algorithms. The parameters are either related to the algorithm (number of iterations, number of ants, heuristics, satisfiability versus optimization) or to the pheromones (initialization value, evaporation rate, pheromones versus heuristic information, simu-finite related parameters).
We consider in our experiments the following parameters:

- nbIte is the maximum number of iterations,
- nbAnts is the number of artificial ants,
- pMax is the maximum value of a pheromone increase,
- pMin is the minimum value of a pheromone increase,
- ρ is the evaporation rate,
- evaS is a boolean parameter deciding between total evaporation or restricted evaporation,
- ObjBV,ObjMin,ObjMax are respectively the initialization, minimum and maximum values for relations targets choices,
- RelBV,RelMin,RelMax are respectively the initialization, minimum and maximum values for relation's cardinality choices,
- AttBV,AttMin,AttMax are respectively the initialization, minimum and maximum values for attributes value choices,

- ClassBV,ClassMin,ClassMax are respectively the initialization, minimum and maximum values for final classes choices,
- NOBV is the initialization value for the creation choice of relations targets simufinite sets.

A particular issue in choosing parameters for configuration is the overall size of the solution. With a fast size increase, the algorithm may *miss* the optimal solution. But with a slow increase and a problem for which the solution has a large number of objects, the iterations spent on small size solutions yield an overall larger computation time. As the configuration space may be infinite, we should rely on parameters which offer a convergence to the solution's size.
Furthermore, this overall size may not increase at the same rate as the total number of objects that have been created from the beginning, raising another issue. For instance consider that we favor the creation of objects with a high value for the parameter NOBV. If the pheromones on relations cardinalities favor small solutions, then a given object will statistically be selected less often. Therefore there are fewer iterations dedicated to finding the optimal values for the object. This particular problem does not occur in static problems like CSPs where the number of variables is defined.
Those issues are recurrent problems in ACO algorithms related to exploration versus intensification. It is even more prominent in configuration because it applies in the same time to a solution's size (with optional and dynamic variables), and to its participating objects local instantiations (attributes and classification, which are similar to a CSP assignment).
The increased number of parameters in ACO for configuration is also an issue and finding the best set calls for advanced techniques. We propose in the following to use Particle Swarm Optimization.

Using Particle Swarm Optimization for finding the best parameters

Introduction to Particle Swarm Optimization Particle Swarm Optimization (PSO) is a population based stochastic optimization technique. It was first developed by Dr. Eberhart and Dr. Kennedy in 1995.
A *PSO problem* is defined by an objective function on a multi-dimensional space $f : \mathbb{R}^m \rightarrow \mathbb{R}$, with $[min_j, max_j]$, $0 <= j < m$, being the domains bounds for each dimension j. There are n particles p_i, $0 <= i < n$. Each *particle* represents a potential solution and is defined by a *position* $x_i \in \mathbb{R}^m$ and a *velocity* $v_i \in \mathbb{R}^m$. The position is a potential solution. The velocity is the current direction of a particle in the problem space. The main idea as that particles move through the problem space by following the current optimum particles. Each particle has the knowledge of its own best achieved position and its global's (or neighborhood's) best.
Table 6.6 presents a PSO algorithm. ω is an inertial constant, commonly set to slightly less than 1 and often decreasing over time. $c1$ and $c2$ are constants controlling how much the particle is directed towards optimum positions. They are called a "cognitive"

and a "social" component, respectively, in that they affect how much the particle's best and the global's best influence its movement. Usual values are $c1 = c2 = 2$. $r1$ and $r2$ are random numbers in $[0, 1]$. $x_{i,j}$ is frequently initialized randomly, whereas $v_{i,j}$ is initialized to 0. *terminationConditions* are usually a given number of iterations.
There are many PSO variants, in particular for managing discrete domains in each dimension.

```
procedure pso
  for i := 0 to n do
    initialize x_i and v_i
  initialize x̂_i and ĝ
  while terminationConditionsNotMet do
    for i := 0 to n do
      for j := 0 to m do
        v_{i,j} = ω * v_{i,j} + c1 * r1 * (x̂_{i,j} − x_{i,j}) + c2 * r2 * (ĝ_j − x_{i,j})
        x_{i,j} = x_{i,j} + v_{i,j} if ( f(x_{i,j}) > f(x̂_{i,j}) ) then x̂_{i,j} = x_{i,j}
        if ( f(x_{i,j}) > f(ĝ_j) ) then ĝ_j = x_{i,j}
  return
```

Table 6.6: A **pso algorithm**

PSO for ACO parameters We use a straightforward implementation of PSO for finding the best $ACOC_{class}$ and $ACOC_{graph}$ parameters. Each parameter is a dimension of the PSO problem. Domains of parameters may be either discrete (booleans, integers) or continuous (floats). The objective function is a combination of the solution's quality, number of iterations required to construct it, and overall computation time.
An example of a PSO execution trace on our ACO framework is given in Annex IV.

6.3.4 Experimental results and analysis

We present experiments on randomly generated problems and on a known benchmark of the configuration literature (the rack problem). Each time we used our context implementation of the PSO algorithm (with 20 particles) to discover the best set of parameters for $ACOC$ variants. In the following experiments, nbIte, nbAnts, NOBV, and all max values parameters have been fixed whereas *rho*, evaS, all base and min values, pMin and pMax could vary with each particle.

Random problems We generated the random problems with the following parameters: number of classes, relations density (probability to have a relation between two classes) and relations cardinality difficulty (average distance between minimum and maximum cardinalities). There are no additional constraints, no classes inheritance, and

$ACOC_{class}$ parameters								
nbIte	nbAnts	ρ	evaS	pMin	pMax	objBV	objMin	objMax
200	30	4	restricted	0	39	100	30	90
attBV	attMin	attMax	relBV	relMin	relMax	classBV	classMin	classMax
n/a	n/a	n/a	100	30	90	n/a	n/a	n/a

$ACOC_{class}$ results			
solution found (%)	number of iterations	size	time
100	14	13	177

$ACOC_{graph}$ parameters								
nbIte	nbAnts	ρ	evaS	pMin	pMax	noBV	objMin	objMax
200	30	4	restricted	0	9	100	2	90
attBV	attMin	attMax	relBV	relMin	relMax	classBV	classMin	classMax
n/a	n/a	n/a	100	12	90	n/a	n/a	n/a

$ACOC_{graph}$ results			
solution found (%)	number of iterations	size	time
100	8	13	89

Table 6.7: $ACOC$ experiments on random problems, times in milli-secs

problems are created such that a finite solution exists. We consider the problem of finding a solution without any optimization function. We generated 10 problems with uniform randomness parameters. Each particle solves all of them 200 times with the same set of parameters. The average results are considered. Table 6.7 shows the experimental results obtained by the best particle over 50 PSO iterations.

Rack problem The rack problem is an optimization benchmark for configuration[1], involving cards plugged to racks. The optimization function is the minimization of the overall cost (sum of the rack's prices). The benchmark consists of 4 instances with a fixed number of cards to be plugged.
We only present $ACOC_{graph}$ on the rack problem since $ACOC_{class}$ did not provide satisfying results. Each particle runs on the 4 instances 200 times with the same set of parameters. The average results are considered. Table 6.8 shows the experimental results obtained with the best set of parameters after 50 PSO iterations. Our ACO approach is compared to the isomorphism-free procedure presented in Chapter 5, and the results presented in [60] ([60] are, to the best of our knowledge, the most recent

[1] http://www.csplib.org, problem 31

$ACOC_{graph}$ parameters								
nbIte	nbAnts	ρ	evaS	pMin	pMax	objBV	objMin	objMax
500	30	2	restricted	0	11	100	0	90
attBV	attMin	attMax	relBV	relMin	relMax	classBV	classMin	classMax
100	5	90	100	13	90	100	8	90

instance		$ACOC_{graph}$						ISO	[60]
	optimal solution	solution found (%)	optimal found (%)	average price	nbIterations	size	time	time	time
1	550	100	88	632	79	21	2113	51	340
2	1100	100	37	1521	240	43	6753	36000	3700
3	1200	100	18	1886	301	56	10690	66	45000
4	1150	97	12	1643	249	33	4729	1800	/

Table 6.8: $ACOC_{graph}$ experiments on the rack problems, times in milli-secs

experimental results on the rack problem)[2].

Experimental analysis

We must first say that those experiments are limited and can only be considered preliminary results since many parameters have been fixed and only a limited number of variants of the algorithms have been tested. In particular, there are many known improvements to ACO algorithms that may be applied for configuration. We will discuss these in the next paragraph.
However we were first interested in a usefulness validation. Indeed, using a stochastic approach is new in configuration thus we need to study its potential prior to extensive experiments. Furthermore, the large parameter space of the tool calls for a deep and careful analysis of their effects.
About parameters, we can first observe that PSO converges towards maximum for "base values" (RelBV, ObjBV, ClassBV, AttBV). The benefits of this maximum initialization was already observed in several previous work on ACO. Our intuition about the need for a restricted evaporation is also confirmed in these experiments. We may also note that $ACOC_{graph}$ best sets of parameters are very close in both the random and the rack problems.
We also observe significant differences between parameters of the two variants. The minimum values of $ACOC_{graph}$ are smaller, as well as the maximum pheromone increase. This may indicate that $ACOC_{graph}$ has a more reliable convergence behaviour, whereas $ACOC_{class}$ needs to quickly focus on good instances.
The results on random problems show that the approach can solve simple problems

[2]Note that obviously the two exhaustive methods always find the optimal solution

efficiently. Only few iterations are required to find a solution. They also show a slight advantage to $ACOC_{graph}$ variant. Furthermore, $ACOC_{class}$ has not been able to find any optimal solutions on the rack problems. Analyzing the reasons is a topic of future work.
The $ACOC_{graph}$ results on the rack problem are contrasted. On the one hand, the ACO approach allows to find correct solutions in acceptable computation times. On the other hand, the instances are in average far from the optimal solution, and the optimal solution is not found sufficiently often for a reliable application. Furthermore, a large number of iterations remains after the best solution is found where we are unable to improve the solution. However, we may suppose that the addition of heuristics will help in these directions.

Algorithms perspectives

Several methods have been developed to improve the results of ACO algorithms, and most of them can be applied to configuration. Besides the obvious need for heuristics, we present some of the considered variants.
Concerning the pheromones update, it is possible to rate the value of the current solution by comparing its quality to the best solution found in previous iterations. One of the expected effects is to exit from local optimums with an induced explorative behaviour.
Solution's quality calculus is also central to the pheromones model efficiency. In the presented experiments, all constraints of the model have an equal importance though "for-all" constraints usually apply to several objects. We currently investigate a variant where the number objects affected by a constraint is taken into account. In the case of optimization, we may "fine-tune" the relative importance of constraints satisfiability versus the optimization function, and this relation may evolve once a solution is found.
Another efficient algorithm enhancement is to perform a random restart of the procedure after a given number of iterations. This restart can be triggered when there is no increase in the solutions quality for a certain number of iterations. In the special case of configuration, we can imagine, depending on the problem, a restart triggered when a given upper bound on the instance size is reached, or when the algorithm is "stuck" at the same size for too long.
We may also consider the creation of objects by the algorithm. Currently, the associated pheromones are handled in a very straightforward manner, by comparing the current instance to the one obtained in the previous iteration. Our experiments have shown that slight changes to its update calculus have a large effect on the behaviour: either few objects are created and the satisfiability is poor, or new objects are created so often that intensification is not sufficient. Furthermore, this rate is completely independent from the configured instances size. We believe this point is a crucial issue to the efficiency of ACO for configuration and are considering different supervision enhancements.

Experiments on size evolution

Since we pointed out the original issue of instances size in configuration, we ran a sery of experiments that focuses on its evolution at each ACO iteration. Again, each problem is solved 200 times and average size is considered. We used the best set of parameters discovered in the previous experiments, and we let the solving continue even after a solution was found.

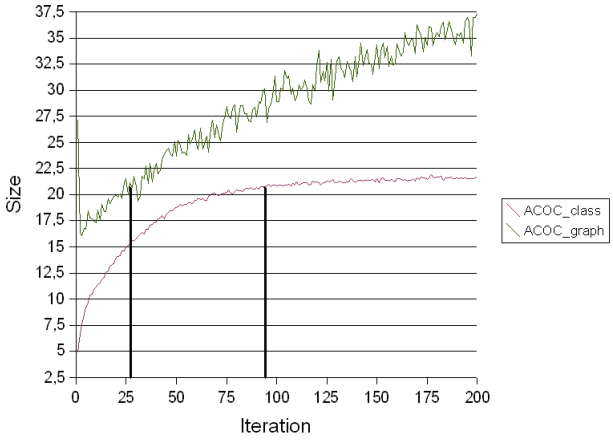


Figure 6.3: Instances size evolution on a random problem

Random problem The results on one random problem are shown in Figure 6.3. Given the results obtained in the previous experiments, we used the randomness parameters which yielded the "hardest" problem. The bottom line corresponds to $ACOC_{class}$ whereas the top line corresponds to $ACOC_{graph}$. A vertical line marks the first time a solution is found.
We can observe that the algorithms behaviours are totally different. $ACOC_{class}$, driven by its (artificial) root relation to classes, increases slowly the size of the problem until a solution is found and then stabilizes. Note that many additional solutions are found in the remaining iterations.
With $ACOC_{graph}$, the size increase is persistent although a solution is found much sooner. However we must say that many solutions are found at greater sizes, which is a particularity of those random problems and may explain this behaviour.

Rack problems The results on the rack problems are displayed in Figure 6.4. Only $ACOC_{graph}$ is tested. A vertical line marks the first time the best solution is found.
We observe that the size evolution is restricted for all problems. Since the problems require a fixed number of cards and only allow up to 5 racks, this behaviour denotes that pheromones correctly drive the ants with respect to the problems constraints. However,

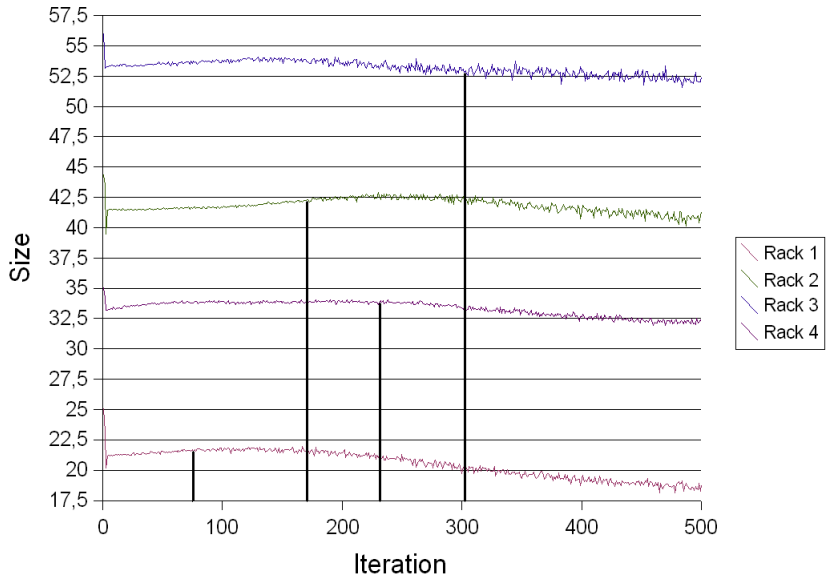


Figure 6.4: Instances size evolution on the rack problems

although the optimal solution is seldom found, the best solution is not improved anymore after a certain number of iterations. This may be correlated to the slight but persistent decrease observed at the same time.

6.4 Conclusion

The application of the ACO meta-heuristic to configuration opens the path for a large field of research. Therefore the work presented here can only be considered as preliminary results. Indeed, the perspectives are wide and additional experiments are required. In particular, we need to analyse the algorithms behaviour with respect to objects creation, as well as experiment various improvement technics.
In particular, the presented implementation suffers from the lack of variable and value heuristics and we may suppose their addition will significantly boost the performances. Realizing a hybrid algorithm using local search to enhance ants solutions is also promising, considering its known positive impact in ACO for CSPs.
Despite its preliminary nature, the presented work already points out the main challenges in ACO for configuration and defines a method which takes into account the whole generality of configuration problems. The experimental results offer a viable starting point for using stochastic procedures in configuration.
Since the proposed framework includes potential solutions to issues raised by first-order logic in ACO algorithms, our theoretical results may be reused in other AI approaches than configuration.
Finally, the abstract library developed for the implementation of constrained object

models is independent from the solving procedure. A practical perspective is to define a modular translation from Z to this library. The resulting combination may then be reused by the configuration community to develop and compare different solving algorithms. In particular, we plan to reuse the library to implement an enumerative algorithm which allows to use our isomorphism rejection procedure in complex configuration problems.

Chapter 7

Conclusion

In this thesis we applied finite model search in the context of constrained object models expressed using the Z language. The combination of constraint programming, description logics and first-order or higher theories allows to model many AI problems in a fully declarative way. The presented formalism is also independent from the search procedure. As solving takes place directly at this high-level language, there is no need for translations and the results can be readily exploited.
We presented an application to SWS composition which illustrates these advantages. The described framework operates at both the abstract level of capabilities and at the concrete level of workflows. At each level, data ontologies are part of the model and can be reasoned about without calling on additional formalisms. In the workflow model, a dual standpoint syntax/execution is combined. In the composition goal model, an original point of view is adopted with the configuration of composition requests. We also raised original issues in SWS composition like automatic extraction of the composite SWS orchestration and choreography with respect to conformance.
The application is validated through experimental results by its integration in an existing SWS framework. Computed orchestrations can be directly executed by a workflow engine. Compared to existing approaches, the described configuration-based composer is competitive in both the features supported and the computational complexity. However scalability issues are encountered when dealing with large input sets.
Indeed, the enumerative nature of finite model search induces a combinatorial explosion that is well known in the field of CSPs. Many methods and algorithms have been developed to improve CSPs scalability. Although configuration brings its own original challenges, it shares many similarities with CSPs so that applying the same kind of methods can be considered.
In this respect, we described an isomorphism rejection method for enumerative configuration algorithms. The presented work, addressing the symmetry issue for dynamically created structures, generalizes to DAGs an existing work on trees. The procedure uses a pseudo-linear algorithm to prevent from creating some isomorphic configurations. The theoretical results are promising, but have not yet been applied to a concrete configuration problem.

We also presented a stochastic uncomplete method for finite model search. To the best of our knowledge, this intense field of research in constraint programming had not yet been considered for first-order theories. The presented approach, based on previous work about ant colonies behaviour, deals with a number of original issues raised by configuration. We obtained promising preliminary results but the field clearly offers many perspectives for improvement.
We believe to have contributed to the generic use of configuration in AI. As opposed to its limited use in traditional industrial applications, we tried to generalize configuration to a more general context wich combines truely dynamic structures and symbolic reasoning with solving efficiency.

Perspectives

Finite model search for constrained object models is a powerful logical paradigm, and the perspectives in both its theoretical and application aspects are wide.
The presented application framework for SWS composition still has a number of limitations and uncovered features. Modelling an extended coverage of workflow patterns, as well as compensation requirements, may reveal unknown issues in configuration's expressive power. The executability of the configured workflows can also be extended. For instance an improvement in dead-locks prevention, or an improved model of token-flow taking into account tokens multiplicity and iterative executions of the workflow.
There are numerous applications to configuration that have not yet been explored. Moreover, the result of configuration is itself a model that can be used as a configuration model for another problem. This opens perspectives for multi-purpose architectures. For instance, [28] proposes to extract semantics of descriptive texts through the creation of a model of the world-knowledge. If we consider these world-knowledge descriptions as ontologies, the result can be seen as a natural language query for the composer or any configuration-based reasoning tool.
From the theoretical point of view, reasoning at the meta-model level is a perspective. Although it implies a significant change in configuration's fundamentals, and thus modelling and solving issues, the resulting expressive potential deserves interest.
As for other AI fields, the practical use of configuration depends on its computational potential. However until recently finite model search attracted little research interest in the AI community. The presented work at the operational level has many directions for future work.
For the isomorphism rejection procedure, other linear-time methods able to remove isomorphic structures are currently investigated, as well as the exploitation of detected symmetries during objects instantiation. In order to apply it to concrete configuration scenarios, we plan to implement it in an enumerative search algorithm using the java abstract library of constrained object models developed for the stochastic approach.
For the application of ACO to configuration, there are several directions currently investigated. First we will add heuristics to balance the pheromones during the search. We also consider to further analyse the experiments with respect to the solution's size, and to diversify ants behaviour with exploration and intensification parts of the colony.

Interleaving local search is also a promising perspective.
Finally, we wish to further advocate the use of Z for constrained object models as a common formalism, from which different solving algorithms can be compared. In that respect, a modular translation from Z based COMs to our abstract library is a topic of future work. The combination Z-COMs/solvers would make of the formalism an AI language with executable finite model semantics.

Annexes

Annex I: Fragments of the AD-S implementation in JConfigurator

In this annex we present an overview of the AD-S COM implementation in ILOG's tool JConfigurator. In JConfigurator, a COM is expressed as an object model together with constraints.

Object model

JConfigurator provides specifications for object models that are close, but not standard, UML class diagrams. Figure 7.1 shows a fragment of the AD-S object model using the tool's graphical interface.
We may note that only oriented relations are allowed, i.e only roles can be implemented. Thus general relations are specified as follows:

- for a relation where two roles are specified, we need to include the reversal property which is directly available in ILOG's modelisation tool,
- for an association relation, both roles are specified,
- for a composition relation, we need to include that the role from the source class to the target class is *exclusive*. An exclusive role means that the target object cannot be shared with another source, thus well expressing the semantics of composition. Again, the exclusive property is an option of the modelisation tool.

Furthermore, all relations have an upper bound to their cardinality. This restriction to the general configuration context requires to carefully decide on appropriate upper bounds with respect to the current problem.
We also point out the special relation called *classField* used for the *concept* relations. As already discussed in Chapter 4, this original property brought by JConfigurator's specifies that only the target class is to be decided, whereas the target object should not be configured. In the AD-S model, we use it for "concept" relations.

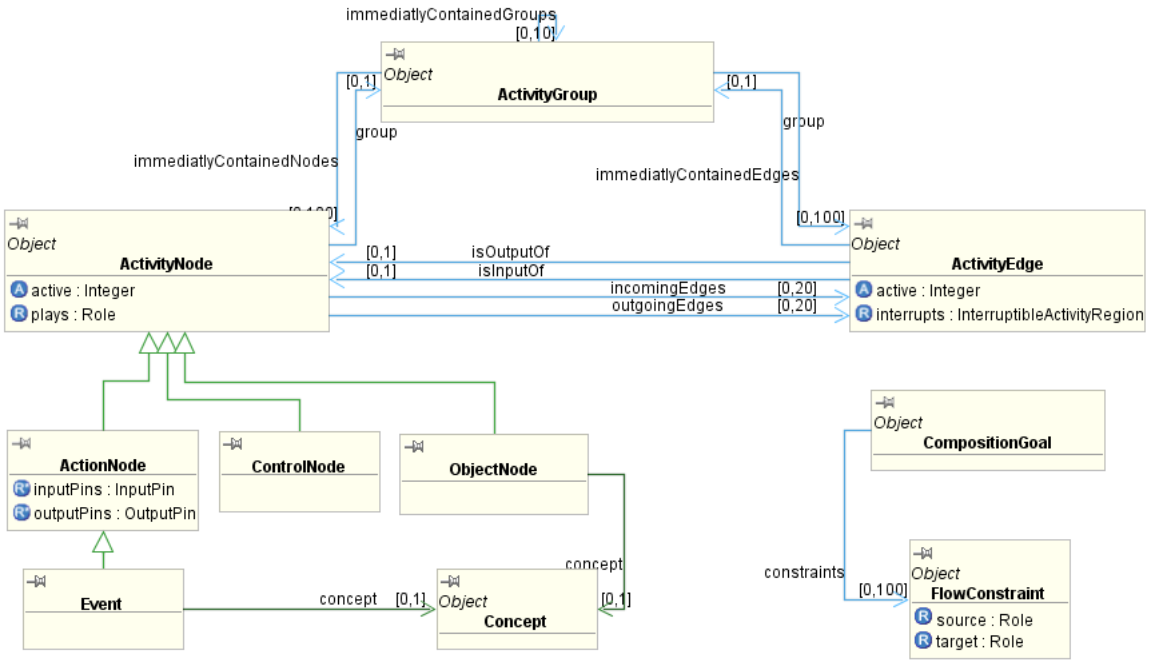


Figure 7.1: Fragment of the AD-S object model in JConfigurator

Constraints

Additional constraints are specified in a separate file. JConfigurator provides a java library allowing to express constraints on the object model. In order to place generic constraints on model elements, it is required to create *logical variables* that represent objects of a given class in the current instance. The syntax is as follows:

```
IloClass classVar = om.getClass("ObjectModelClassName");
IloClass logicVar = om.logicObjectVar(classVar);
```

In the following we present fragments of the AD-S constraints in JConfigurator. We give, for each main type of constraints, some examples of how they are implemented. The declaration of logic variables is omitted as names are self explanatory.

```
//********************
// Predefined groups
//********************

// User Activity Group
UG = om.makeInstance(GeneralGroupClass,"UG");
// Orchestration Activity Group
OG = om.makeInstance(GeneralGroupClass,"OG");
```

```
//*************************
// Cardinality constraints
//*************************

// Control Nodes
//---------------

// AbstractJoin: 1 outgoing edge
om.add(om.forAll(AbstractJoinNodeVar,
    om.eq(AbstractJoinNodeVar.getIntField("active"),1),
    om.eq(om.cardinality(AbstractJoinNodeVar.getObjectSetField("outgoingEdges")),1)
));

// AbstractJoin: at least 1 incoming edge
om.add(om.forAll(AbstractJoinNodeVar,
    om.eq(AbstractJoinNodeVar.getIntField("active"),1),
    om.ge(om.cardinality(AbstractJoinNodeVar.getObjectSetField("incomingEdges")),1)
));

// Final Node: 1 incoming edge, no outcoming edge
om.add(om.forAll(FinalNodeVar,
    om.eq(FinalNodeVar.getIntField("active"),1),
    om.and(
        om.eq(om.cardinality(FinalNodeVar.getObjectSetField("outgoingEdges")),0),
        om.eq(om.cardinality(FinalNodeVar.getObjectSetField("incomingEdges")),1)
    )
));

// Action Nodes
//--------------

// User Send Events: no pins, no edges
om.add(om.forAll(SendEventVar,
    om.and(
        om.eq(SendEventVar.getIntField("active"),1),
        om.eq(SendEventVar.getObjectField("group"),UG)
    ),
    om.and(
        om.and(
            om.eq(om.cardinality(SendEventVar.getObjectSetField("outputPins")),0),
            om.eq(om.cardinality(SendEventVar.getObjectSetField("inputPins")),0)
        ),
        om.and(
            om.eq(om.cardinality(SendEventVar.getObjectSetField("outgoingEdges")),0),
            om.eq(om.cardinality(SendEventVar.getObjectSetField("incomingEdges")),0)
        )
    )
));
```

```
// Object Nodes
//--------------

// OutputPin: no incoming edge, 1 outgoing edge, 1 node
om.add(om.forAll(OutputPinVar,
    om.eq(OutputPinVar.getIntField("active"),1),
    om.and(
        om.eq(om.cardinality(OutputPinVar.getObjectField("node")),1),
        om.and(
            om.eq(om.cardinality(OutputPinVar.getObjectSetField("outgoingEdges")),1),
            om.eq(om.cardinality(OutputPinVar.getObjectSetField("incomingEdges")),0)
        )
    )
));

//*********************************
// Activity propagation constraints
//*********************************

// Activity Edges
//----------------

// Activity Edges: isOutputOf is active
om.add(om.forAll(ActivityEdgeVar,
    om.eq(ActivityEdgeVar.getIntField("active"),1),
    om.and(
        om.eq(om.cardinality(ActivityEdgeVar.getObjectField("isOutputOf")),1),
        om.eq(ActivityEdgeVar.getObjectField("isOutputOf").getIntField("active"),1)
    )
));

// Object Nodes
//--------------

// InputPin : 1 incomingEdge which is active
om.add(om.forAll(InputPinVar,
    om.eq(InputPinVar.getIntField("active"),1),
    om.eq(om.sum(InputPinVar.getObjectSetField("incomingEdges"), "active"),1)
));

// OutputPin : node is active
om.add(om.forAll(OutputPinVar,
    om.eq(OutputPinVar.getIntField("active"),1),
    om.eq(OutputPinVar.getObjectField("node").getIntField("active"),1)
));
```

```
// Action Nodes
//--------------

// Accept Events: partner is active
om.add(om.forAll(AcceptEventVar,
    om.eq(AcceptEventVar.getIntField("active"),1),
    om.and(
        om.eq(om.cardinality(AcceptEventVar.getObjectField("partner")),1),
        om.eq(AcceptEventVar.getObjectField("partner").getIntField("active"),1)
    )
));

// Action Nodes: all inputPins and incomingEdges are active
om.add(om.forAll(ActionNodeVar,
    om.eq(ActionNodeVar.getIntField("active"),1),
    om.and(
        om.eq(
            om.sum(ActionNodeVar.getObjectSetField("inputPins"), "active"),
            om.cardinality(ActionNodeVar.getObjectSetField("inputPins"))
        ),
        om.eq(
            om.sum(ActionNodeVar.getObjectSetField("incomingEdges"), "active"),
            om.cardinality(ActionNodeVar.getObjectSetField("incomingEdges"))
        )
    )
));

// Control Nodes
//---------------

// JoinNode : all incomingEdges are active
om.add(om.forAll(JoinNodeVar,
    om.eq(JoinNodeVar.getIntField("active"),1),
    om.eq(
        om.sum(JoinNodeVar.getObjectSetField("incomingEdges"), "active"),
        om.cardinality(JoinNodeVar.getObjectSetField("incomingEdges"))
    )
));

// MergeNode : at least one incomingEdge is active
om.add(om.forAll(MergeNodeVar,
    om.eq(MergeNodeVar.getIntField("active"),1),
    om.ge(om.sum(MergeNodeVar.getObjectSetField("incomingEdges"), "active"),1)
));
```

```
//***********************
// Concepts constraints
//***********************

// If an Objectflow is outputOf an object node, it shares the same concept
om.add(om.forAll(ObjectFlowVar,
    om.and(
        om.eq(ObjectFlowVar.getIntField("active"),1),
        om.eq(ObjectFlowVar.getObjectField("isOutputOf")
            .getClassification(),ObjectNodeClass)
    ),
    om.eq(
        om.downCast(ObjectFlowVar.getObjectField("isOutputOf"),ObjectNodeClass)
            .getClassField("concept"),
        ObjectFlowVar.getClassField("concept")
    )
));

// Partner events share the same concept as partner
om.add(om.forAll(AcceptEventVar,
    om.eq(AcceptEventVar.getIntField("active"),1),
    om.eq(
        AcceptEventVar.getClassField("concept"),
        AcceptEventVar.getObjectField("partner").getClassField("concept")
    )
));

//**************************
// Miscellaneous constraints
//**************************

// Partner events have one end in the orchestration group
om.add(om.forAll(AcceptEventVar,
    om.eq(AcceptEventVar.getIntField("active"),1),
    om.or(
        om.eq(OG,AcceptEventVar.getObjectField("group")),
        om.eq(OG,AcceptEventVar.getObjectField("partner").getObjectField("group"))
    )
));
```

```
//*****************************
// CompositionGoal constraints
//*****************************

// ControlFlowConstraints
//----------------------

// An activityEdge exists between
// CF target's playedBy node and CF source's playedBy node
om.add(om.forAll(ControlFlowConstraintVar,
    om.member(
        ControlFlowConstraintVar.getObjectField("source")
            .getObjectField("isPlayedBy"),
        om.objectUnion(
            ControlFlowConstraintVar.getObjectField("target")
                .getObjectField("isPlayedBy").getObjectSetField("incomingEdges"),
            "isOutputOf"
        )
    )
));
```

Annex II: WSML grammar for the AD-S language

```
activity_diagram      = group startnode;

//ACTIVITY GROUPS

group                 = {activitygroup} t_activitygroup id?
                                        activitygroupcontents |
                        {interruptibleregion} t_interruptibleregion id?
                                        activitygroupcontents;

activitygroupcontents = lbrace node+ edge* group* rbrace;

//NODES

node                  = {generalaction} t_generalaction id? nodecontents |
                        {oomediator} t_oomediator id? ref nodecontents |
                        {flowstart} t_flowstart id? |
                        {flowfinal} t_flowfinal id? |
                        {activityfinal} t_activityfinal id? |
                        {aggregation} t_aggregation id? nodecontents |
                        {extraction} t_extraction id? nodecontents |
                        {operation} t_operation id? nodecontents
                                    t_definedBy id |
                        {fork} t_fork id? |
                        {join} t_join id? |
                        {decision} t_decision id? |
                        {merge} t_merge id? |
                        {sendeventaction} t_sendeventaction id? target |
                        {accepteventaction} t_accepteventaction id? source |
                        {objectnode} t_objectnode id?;

startnode             = t_startnode id;

nodecontents          = pin*;

pin                   = {inputpin} t_inputpin id  |
                        {outputpin} t_outputpin id;
```

```
//EDGES

edge                  = {controlflow} t_controlflow id? edgecontents |
                        {dataflow} t_dataflow id? edgecontents;

edgecontents          = source target interrupting? guard?;

interrupting          = t_interrupts id;

guard                 = {else} t_guard t_else |
                        {expression} t_guard log_expr;

//TERMINALS

t_adchoreography      = 'adChoreography';
t_activitygroup       = 'activityGroup';
t_interruptibleregion = 'interruptibleRegion';
t_generalaction       = 'generalAction';
t_admediator          = 'mediator';
t_aggregation         = 'aggregation';
t_flowstart           = 'flowStart';
t_flowfinal           = 'flowFinal';
t_activityfinal       = 'activityFinal';
t_aggregation         = 'aggregation';
t_extraction          = 'extraction';
t_fork                = 'fork';
t_join                = 'join';
t_decision            = 'decision';
t_merge               = 'merge';
t_operation           = 'operation';
t_accepteventaction   = 'acceptEventAction';
t_sendeventaction     = 'sendEventAction';
t_inputpin            = 'inputPin';
t_outputpin           = 'outputPin';
t_controlflow         = 'controlFlow';
t_dataflow            = 'dataFlow';
t_guard               = 'guard';
t_else                = 'else';
t_startnode           = 'startNode';
t_interrupts          = 'interrupts';
t_objectnode          = 'objectNode';
```

Annex III: Constraints Object Models Library Usage Example

Defining the COM

```
public class ConfigModels {

public static CModel rackInstance1() {

// Create a configuration model
CModel cModel = new CModel();

// Create classes

// Root
CMClass rootclass = new CMClass("Root");
cModel.classes.add(rootclass);

// Set as root object
cModel.addRoot(rootclass);

// Racks
CMClass aclass = new CMClass("Rack");
cModel.classes.add(aclass);
aclass.abstractClass = true;

// Add attribute
Integer pValues[] = new Integer[2];
pValues[0] = 150;
pValues[1] = 200;
CMAttribute powerSupplied =
    new CMAttribute("powerSupplied",CMAttributeIntegerValue.getInstance(),
    new CMAttributeDiscreteDomain(pValues));
aclass.attributes.add(powerSupplied);
```

```
CMClass a0class = new CMClass("Rack0");
cModel.classes.add(a0class);

CMClass a1class = new CMClass("Rack1");
cModel.classes.add(a1class);

aclass.addSubClass(a0class);
aclass.addSubClass(a1class);

// Restrict Attribute powerSupplied depending on rack type
Set <Integer> a0RSet = new HashSet<Integer>();
a0RSet.add(150);
a0class.addAttributeRestriction(powerSupplied, a0RSet);

Set <Integer> a1RSet = new HashSet<Integer>();
a1RSet.add(200);
a1class.addAttributeRestriction(powerSupplied, a1RSet);


// Cards
CMClass bclass = new CMClass("Card");
cModel.classes.add(bclass);
bclass.abstractClass = true;

CMClass b0class = new CMClass("Card0");
cModel.classes.add(b0class);

CMClass b1class = new CMClass("Card1");
cModel.classes.add(b1class);

CMClass b2class = new CMClass("Card2");
cModel.classes.add(b2class);

CMClass b3class = new CMClass("Card3");
cModel.classes.add(b3class);

bclass.addSubClass(b0class);
bclass.addSubClass(b1class);
bclass.addSubClass(b2class);
bclass.addSubClass(b3class);

// Add attribute
Integer prValues[] = new Integer[4];
prValues[0] = 20;
```

```
prValues[1] = 40;
prValues[2] = 50;
prValues[3] = 75;
CMAttribute powerRequired =
    new CMAttribute("powerRequired",CMAttributeIntegerValue.getInstance(),
    new CMAttributeDiscreteDomain(prValues));
bclass.attributes.add(powerRequired);

// Restrict Attribute powerRequired depending on card type
Set <Integer> b0RSet = new HashSet<Integer>();
b0RSet.add(20);
b0class.addAttributeRestriction(powerRequired, b0RSet);

Set <Integer> b1RSet = new HashSet<Integer>();
b1RSet.add(40);
b1class.addAttributeRestriction(powerRequired, b1RSet);

Set <Integer> b2RSet = new HashSet<Integer>();
b2RSet.add(50);
b2class.addAttributeRestriction(powerRequired, b2RSet);

Set <Integer> b3RSet = new HashSet<Integer>();
b3RSet.add(75);
b3class.addAttributeRestriction(powerRequired, b3RSet);


// Create relations
rootclass.relations.add(
    new CMRelation(cModel,"root-racks",rootclass,aclass,0,5));
CMRelation rackCards =
    new CMRelation(cModel,"rack-cards",aclass,bclass,0,16,true);
aclass.relations.add(rackCards);
a0class.addRelationRestriction(rackCards, 0, 8);

CMRelation cardRack = new CMRelation(cModel,"card-rack",bclass,aclass,1,1);
bclass.relations.add(cardRack);

rackCards.setReciprocity(cardRack);


// Constraints

// powerSupplied >= powerRequired
CMLogicVar lv = new CMLogicVar(aclass);
```

```
CMConstraintBinaryLeInt c1 =
    new CMConstraintBinaryLeInt(lv.getRelationTargets("rack-cards")
        .getAttributeSum("powerRequired"),
        lv.getAttributeInt("powerSupplied"));
CMConstraintForAll c2 = new CMConstraintForAll(lv,c1);
cModel.constraints.add(c2);
c2.name = "rack power";

// number of Cards to be used for each type
CMLogicClassSet lcsCard0 = new CMLogicClassSet(b0class);
CMConstraintBinaryEqInt card0eq =
    new CMConstraintBinaryEqInt(lcsCard0.getCardinality(),new CMInteger(10));
CMConstraintForAllLogicClassSet card0 =
    new CMConstraintForAllLogicClassSet(lcsCard0,card0eq);
cModel.constraints.add(card0);
card0.name = "10 cards of type 0";

CMLogicClassSet lcsCard1 = new CMLogicClassSet(b1class);
CMConstraintBinaryEqInt card1eq =
    new CMConstraintBinaryEqInt(lcsCard1.getCardinality(),new CMInteger(4));
CMConstraintForAllLogicClassSet card1 =
    new CMConstraintForAllLogicClassSet(lcsCard1,card1eq);
cModel.constraints.add(card1);
card1.name = "4 cards of type 1";

CMLogicClassSet lcsCard2 = new CMLogicClassSet(b2class);
CMConstraintBinaryEqInt card2eq =
    new CMConstraintBinaryEqInt(lcsCard2.getCardinality(),new CMInteger(2));
CMConstraintForAllLogicClassSet card2 =
    new CMConstraintForAllLogicClassSet(lcsCard2,card2eq);
cModel.constraints.add(card2);
card2.name = "2 cards of type 2";

CMLogicClassSet lcsCard3 = new CMLogicClassSet(b3class);
CMConstraintBinaryEqInt card3eq =
    new CMConstraintBinaryEqInt(lcsCard3.getCardinality(),new CMInteger(1));
CMConstraintForAllLogicClassSet card3 =
    new CMConstraintForAllLogicClassSet(lcsCard3,card3eq);
cModel.constraints.add(card3);
card3.name = "1 card of type 3";

return cModel;
}
}
```

Solving with the ACO Engine

```
// Create an ACO-based Configuration engine
AntConfigEngine engine = new AntConfigEngine();

// Set up some engine parameters
engine.useACOCClass = 0;
engine.numberOfAnts = 20;
engine.numberOfIterations = 500;

// Set up some ACO parameters
engine.antEvaporationStyle = 1;
engine.antObjectMax = 90;
engine.antRelationBaseValue = 80;

// Set up the configuration model
CModel cModel = ConfigModels.rackInstance1();
CMInstance instance = engine.solve(cModel);
instance.display();
```

Annex IV: Example of a PSO execution trace for ACO

We present in the next page an example of a PSO execution trace on the rack problem. List of varying parameters used in the PSO execution:

- pMax is the maximum value of a pheromone increase,
- Eva is the evaporation rate (ρ),
- ObjBV,ObjMi are respectively the initialization and minimum values for relations targets choices,
- RelBV,RelMi are respectively the initialization and minimum values for relation's cardinality choices,
- AttBV is the initialization value for attributes value choices,
- PNO is the initialization value for the creation choice of relations targets simu-finite sets.

List of results variables:

- Found%: number of times a solution was found,
- Size: average size of the solutions,
- IteF: average iteration where the solution was found,
- Time: average time in milli-seconds when a solution was found,
- Optim: average optimization value of the returned instances,
- Part.: global best particle number,
- Ite: current PSO iteration.

ACO parameters								Global best particle results					
PMax 1-40	Eva 1-40	NOBV 10-100	RelBV 1-100	ObjBV 1-100	AttBV 1-100	PNO 1-100	ObjMi 0-30	RelMi 0-30	Found %	Size	IteF	Time	Optim	Part.	Ite
24	19	86	70	39	72	82	22	25	82	39	95	6185	1569	11	0
25	19	91	100	69	90	56	13	28	86	39	121	7271	1543	6	1
17	19	93	100	50	100	62	4	17	89	39	86	5192	1515	14	2
4	5	100	99	12	94	53	0	19	94	38	135	5473	1484	6	5
10	1	100	100	10	100	50	0	17	96	38	262	5792	1409	2	6
10	1	10	100	10	100	88	0	16	97	38	266	5810	1417	3	8
10	4	87	100	20	80	58	0	15	98	38	158	6178	1466	16	9
10	3	76	100	10	54	57	0	15	98	38	178	6132	1457	17	11
10	3	87	100	10	16	9	0	13	98	38	184	6199	1446	18	16
10	2	90	100	10	100	11	0	12	98	38	225	6103	1436	1	20
10	1	97	100	10	100	9	0	18	98	38	263	5801	1427	1	21
10	1	100	100	63	84	9	0	16	98	38	272	5801	1416	14	23
10	1	100	100	77	100	9	0	14	99	38	262	5819	1428	2	25
10	2	100	100	48	100	9	0	14	99	38	217	6039	1426	15	39
10	2	100	100	100	100	9	0	12	99	38	217	6103	1424	6	46
10	2	100	100	100	100	9	0	13	99	38	217	6071	1420	8	49

Example of a PSO execution trace

Bibliography

[1] Patrick Albert, Christian de Sainte Marie, Mathias Kleiner, and Laurent Henocque. Specification of the composition prototype. http://dip.semanticweb.org/deliverables.html, December 2005. DIP Deliverable D4.12.

[2] Patrick Albert, Christian de Sainte Marie, Mathias Kleiner, and Laurent Henocque. Composition prototype. http://dip.semanticweb.org/deliverables.html, June 2006. DIP Deliverable D4.15.

[3] Patrick Albert, Laurent Henocque, and Mathias Kleiner. Configuration-based workflow composition. In *ICWS* [54], pages 285–292.

[4] Patrick Albert, Laurent Henocque, and Mathias Kleiner. A constrained object model for configuration based workflow composition. In Christoph Bussler and Armin Haller, editors, *Business Process Management Workshops*, volume 3812, pages 102–115, 2005.

[5] Jérôme Amilhastre, Hélène Fargier, and Pierre Marquis. Consistency restoration and explanations in dynamic csps application to configuration. *Artif. Intell.*, 135(1-2):199–234, 2002.

[6] Matteo Baldoni, Cristina Baroglio, Alberto Martelli, and Viviana Patti. Reasoning about interaction protocols for web service composition. *Electr. Notes Theor. Comput. Sci.*, 105:21–36, 2004.

[7] Virginia E. Barker and Dennis E. O'Connor. Expert systems for configuration at digital: Xcon and beyond. *Commun. ACM*, 32(3):298–318, 1989.

[8] B. Benhamou and L. Henocque. Finite model search for equational theories. In *4th International Conference Artificial Intelligence and Symbolic Computation, AISC'98*, LNCS, pages 84–93, Plattsburgh, USA, 1998. Springer Verlag.

[9] V. Richard Benjamins and Dieter Fensel. Editorial: problem-solving methods. *Int. J. Hum.-Comput. Stud.*, 49(4):305–313, 1998.

[10] Piergiorgio Bertoli and Marco Pistore. Planning with extended goals and partial observability. In Shlomo Zilberstein, Jana Koehler, and Sven Koenig, editors, *ICAPS*, pages 270–278. AAAI, 2004.

[11] S. Bhiri and al. An orchestration and business process ontology. http://dip.semanticweb.org/deliverables.html, January 2005. DIP Deliverable D3.4.

[12] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[13] Ronald J. Brachman and James G. Schmolze. An overview of the kl-one knowledge representation system. *Cognitive Science*, 9(2):171–216, 1985.

[14] G. Brinkmann. Fast generation of cubic graphs. *J. Graph Theory*, 23:139–149, 1996.

[15] Antonio Brogi, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing web service choreographies. *Electr. Notes Theor. Comput. Sci.*, 105:73–94, 2004.

[16] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In Paolo Ciancarini and Herbert Wiklicky, editors, *COORDINATION*, volume 4038 of *Lecture Notes in Computer Science*, pages 63–81. Springer, 2006.

[17] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. An investigation of some properties of an "ant algorithm". In Reinhard Männer and Bernard Manderick, editors, *PPSN*, pages 515–526. Elsevier, 1992.

[18] Ion Constantinescu, Walter Binder, and Boi Faltings. Flexible and efficient matchmaking and ranking in service directories. In *ICWS* [54], pages 5–12.

[19] Ion Constantinescu, Boi Faltings, and Walter Binder. Large scale, type-compatible service composition. In *ICWS* [53], pages 506–513.

[20] Roman Cunis, Andreas Günter, Ingo Syska, Heino Peters, and Heiner Bode. Plakon - an approach to domain-independent construction. In *IEA/AIE (2)*, pages 866–874, 1989.

[21] Adnan Darwiche. New advances in compiling cnf into decomposable negation normal form. In de Mántaras and Saitta [22], pages 328–332.

[22] Ramon López de Mántaras and Lorenza Saitta, editors. *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. IOS Press, 2004.

[23] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artif. Intell.*, 38(3):353–366, 1989.

[24] Lois M. L. Delcambre, Christian Kop, Heinrich C. Mayr, John Mylopoulos, and Oscar Pastor, editors. *Conceptual Modeling - ER 2005, 24th International Conference on Conceptual Modeling, Klagenfurt, Austria, October 24-28, 2005, Proceedings*, volume 3716 of *Lecture Notes in Computer Science*. Springer, 2005.

[25] Marco Dorigo and Christian Blum. Ant colony optimization theory: A survey. *Theor. Comput. Sci.*, 344(2-3):243–278, 2005.

[26] Marco Dorigo, Gianni Di Caro, and Luca Maria Gambardella. Ant algorithms for discrete optimization. *Artificial Life*, 5(2):137–172, 1999.

[27] Marco Dorigo and Luca Maria Gambardella. Ant colony system: a cooperative learning approach to the traveling salesman problem. *IEEE Trans. Evolutionary Computation*, 1(1):53–66, 1997.

[28] M. Estratat. *Vers les grammaires de configuration*. thèse de doctorat, Université Aix-Marseille III, novembre 2006. directeur : Laurent Henocque.

[29] Mathieu Estratat and Laurent Henocque. Parsing languages with a configurator. In de Mántaras and Saitta [22], pages 591–595.

[30] Torsten Fahle, Stefan Shamberger, and Meinhof Sellman. Symmetry breaking. In *Proceedings of CP'01*, pages 93–107, 2001.

[31] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Configuration knowledge representation using uml/ocl. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 49–62. Springer-Verlag, 2002.

[32] Dieter Fensel, Enrico Motta, Frank van Harmelen, V. Richard Benjamins, Monica Crubézy, Stefan Decker, Mauro Gaspari, Rix Groenboom, William E. Grosso, Mark A. Musen, Enric Plaza, Guus Schreiber, Rudi Studer, and Bob J. Wielinga. The unified problem-solving method development language upml. *Knowl. Inf. Syst.*, 5(1):83–131, 2003.

[33] G. Fleischanderl, G. Friedrich, A. Haselböck, H. Schreiner, and M. Stumptner. Configuring large-scale systems with generative constraint satisfaction. *IEEE Intelligent Systems - Special issue on Configuration*, 13(7), 1998.

[34] Pierre Flener, Justin Pearson, Meinoff Sellman, and Pascal Van Hentenrick. Static and dynamic structural symmetry breaking. In *proceedings of Principles and Practice of Constraint Programming - CP 2006*, pages 695–699, Nantes, France, 2006. Springer.

[35] Filippo Focacci and Michela Milano. Global cut framework for removing symmetries. In *proceedings of Principles and Practice of Constraint Programming - CP 2001*, pages 77–92, Paphos, Cyprus, 2001. Springer.

[36] G. Friedrich and M. Stumptner. Consistency-based configuration. In *AAAI-99 Workshop on configuration*, pages 35–40, 1999.

[37] S. Galizia and al. An ontology for web service choreography. http://dip.semanticweb.org/documents/D3-5.pdf, January 2005. DIP Deliverable D3.5.

[38] Luca Maria Gambardella and Marco Dorigo. Solving symmetric and asymmetric tsps by ant colonies. In *International Conference on Evolutionary Computation*, pages 622–627, 1996.

[39] Ian Gent, Warwick Harvey, Tom Keysley, and Steve Linton. Generic sbdd using computational group theory. In *proceedings of Principles and Practice of Constraint Programming - CP 2003*, pages 333–347, Kinsale, Ireland, 2003. Springer.

[40] Ian P. Gent and Barbara M. Smith. Symmetry Breaking in Constraint Programming. In Werner Horn, editor, *Proceedings of ECAI 2000*, pages 599–603. IOS Press, 2000.

[41] Shahram Ghandeharizadeh, Craig A. Knoblock, Christos Papadopoulos, Cyrus Shahabi, Esam Alwagait, José Luis Ambite, Min Cai, Ching-Chien Chen, Parikshit Pol, Rolfe R. Schmidt, Saihong Song, Snehal Thakkar, and Runfang Zhou. Proteus: A system for dynamically composing and intelligently executing web services. In Liang-Jie Zhang, editor, *ICWS*, pages 17–21. CSREA Press, 2003.

[42] Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[43] Asunción Gómez-Pérez, Rafael González-Cabero, and Manuel Lama. Ode sws: A framework for designing and composing semantic web services. *IEEE Intelligent Systems*, 19(4):24–31, 2004.

[44] Stéphane Grandcolas, Laurent Henocque, and Nicolas Prcovic. A canonicity test for configuration. In Francesca Rossi, editor, *CP*, volume 2833 of *Lecture Notes in Computer Science*, pages 853–857. Springer, 2003.

[45] Object Management Group. *UML v. 2.0 specification*. OMG, 2003.

[46] Thomas R. Gruber, Gregory R. Olsen, and Jay T. Runkel. The configuration design ontologies and the vt elevator domain theory. *Int. J. Hum.-Comput. Stud.*, 44(3-4):569–598, 1996.

[47] Andreas Günter and Christian Kühn. Knowledge-based configuration: Survey and future directions. In Frank Puppe, editor, *XPS*, volume 1570 of *Lecture Notes in Computer Science*, pages 47–66. Springer, 1999.

[48] Albert Haag. Sales configuration in business processes. *IEEE Intelligent Systems*, 13(4):78–85, 1998.

[49] Laurent Henocque. Modeling object oriented constraint programs in Z. *RACSAM (Revista de la Real Academia De Ciencias serie A Mathematicas), special issue about Artificial Intelligence and Symbolic Computing*, pages 127–152, 2004.

[50] Laurent Henocque and Mathias Kleiner. *Composition: Combining Web Service Functionality in Composite Orchestrations*, chapter Composition, pages 245–286. Springer Berlin Heidelberg, 2007.

[51] Laurent Henocque and Nicolas Prcovic. Practically handling some configuration isomorphisms. In *ICTAI*, pages 90–97. IEEE Computer Society, 2004.

[52] Ian Horrocks and Peter F. Patel-Schneider. Reducing owl entailment to description logic satisfiability. *J. Web Sem.*, 1(4):345–357, 2004.

[53] IEEE. *Proceedings of the IEEE International Conference on Web Services (ICWS'04), June 6-9, 2004, San Diego, California, USA*. IEEE Computer Society, 2004.

[54] IEEE. *2005 IEEE International Conference on Web Services (ICWS 2005), 11-15 July 2005, Orlando, FL, USA*. IEEE Computer Society, 2005.

[55] Rune M. Jensen. Clab: A c++ library for fast backtrack-free interactive product configuration. In Mark Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, page 816. Springer, 2004.

[56] Werner E. Juengst and Michael Heinrich. Using resource balancing to configure modular systems. *IEEE Intelligent Systems*, 13(4):50–58, 1998.

[57] Ulrich Junker. Configuration. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 13. Elsevier, 2006.

[58] Ulrich Junker and Daniel Mailharro. The logic of ilog (j)configurator: Combining constraint programming with a description logic. In *proceedings of Workshop on Configuration, IJCAI'03*, 2003.

[59] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *J. ACM*, 42(4):741–843, 1995.

[60] Z. Kiziltan and B. Hnich. Symmetry breaking in a rack configuration problem. In *Proc. of the IJCAI'01 Workshop on Modelling and Solving Problems with Constraints, Seattle*, 2001.

[61] Jens Lemcke, Barry Norton, Carlos Pedrinaci, Mathias Kleiner, and Laurent Henocque. Ontology for web services choreography and orchestration v2. http://dip.semanticweb.org/deliverables.html, July 2006. DIP Deliverable D3.9.

[62] Hector J. Levesque, Fiora Pirri, and Raymond Reiter. Foundations for the situation calculus. *Electron. Trans. Artif. Intell.*, 2:159–178, 1998.

[63] E. M. Luks. Isomorphism of graphs of bounded valence can be tested in polynomial time. *J. Comput. System Sci.*, 25:42–49, 1982.

[64] Diego Magro and Pietro Torasso. Decomposition strategies for configuration problems. *AI EDAM*, 17(1):51–73, 2003.

[65] D. Mailharro. A classification and constraint based framework for configuration. *AI-EDAM : Special issue on Configuration*, 12(4):383 – 397, 1998.

[66] M.Bravetti and G. Zavattaro. A theory for strong service compliance. In *Coordination Models and Languages, 8th International Conference*, 2007.

[67] John P. McDermott. R1: A rule-based configurer of computer systems. *Artif. Intell.*, 19(1):39–88, 1982.

[68] Deborah L. McGuinness and Jon R. Wright. Conceptual modelling for configuration: A description logic-based approach. *AI EDAM*, 12(4):333–344, 1998.

[69] Sheila A. McIlraith and Tran Cao Son. Adapting golog for composition of semantic web services. In Dieter Fensel, Fausto Giunchiglia, Deborah L. McGuinness, and Mary-Anne Williams, editors, *KR*, pages 482–496. Morgan Kaufmann, 2002.

[70] B. D. McKay. Isomorph-free exhaustive generation. *J. Algorithms*, 26(2):306–324, 1998.

[71] Brendan D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[72] Robin Milner. Operational and algebraic semantics of concurrent processes. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1201–1242. 1990.

[73] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[74] Sanjay Mittal and Brian Falkenhainer. Dynamic constraint satisfaction problems. In *AAAI*, pages 25–32, 1990.

[75] Sanjay Mittal and Felix Frayman. Towards a generic model of configuraton tasks. In *IJCAI*, pages 1395–1401, 1989.

[76] B. Nebel. Reasoning and revision in hybrid representation systems. *Lecture Notes in Artificial Intelligence*, 422, 1990.

[77] Barry Norton, Simon Foster, and Andrew Hughes. A compositional operational semantics for owl-s. In Mario Bravetti, Leïla Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *Lecture Notes in Computer Science*, pages 303–317. Springer, 2005.

[78] Barry Norton and Carlos Pedrinaci. 3-level service composition and cashew: A model for orchestration and choreography in semantic web services. In Robert Meersman, Zahir Tari, and Pilar Herrero, editors, *OTM Workshops (1)*, volume 4277 of *Lecture Notes in Computer Science*, pages 58–67. Springer, 2006.

[79] Gary J. Nutt. Book review: Coloured petri nets, basic concepts, analysis methods and practical use by kurt jensen. *Operating Systems Review*, 28(1):1–2, 1994.

[80] Patrick Olivier, K. Nakata, M. Landon, and A. McManus. Analogical representations for mechanism synthesis. In Wolfgang Wahlster, editor, *ECAI*, pages 506–510. John Wiley and Sons, Chichester, 1996.

[81] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia P. Sycara. Semantic matching of web services capabilities. In Ian Horrocks and James A. Hendler, editors, *International Semantic Web Conference*, volume 2342 of *Lecture Notes in Computer Science*, pages 333–347. Springer, 2002.

[82] Bernard Pargamin. Extending cluster tree compilation with non-boolean variables in product configuration. *IJCAI-03 Workshop on Configuration*, pages 32–37, 2003.

[83] Marco Pistore, Pierluigi Roberti, and Paolo Traverso. Process-level composition of executable web services: "on-the-fly" versus "once-for-all" composition. In Asunción Gómez-Pérez and Jérôme Euzenat, editors, *ESWC*, volume 3532 of *Lecture Notes in Computer Science*, pages 62–77. Springer, 2005.

[84] Shankar Ponnekanti and Armando Fox. Application-service interoperation without standardized service interfaces. In *PerCom*, pages 30–. IEEE Computer Society, 2003.

[85] Jean-François Puget. Dynamic lex constraints. In *proceedings of Principles and Practice of Constraint Programming - CP 2006*, pages 453–467, Nantes, France, 2006. Springer.

[86] Jinghai Rao, Peep Küngas, and Mihhail Matskin. Logic-based web services composition: From service description to process model. In *ICWS* [53], pages 446–453.

[87] Ronald C. Read. Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations. *Annals of Discrete Mathematics*, 2:107–120, 1978.

[88] Nick Russell, Arthur H. M. ter Hofstede, David Edmond, and Wil M. P. van der Aalst. Workflow data patterns: Identification, representation and tool support. In Delcambre et al. [24], pages 353–368.

[89] Nick Russell, Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In Oscar Pastor and João Falcão e Cunha, editors, *CAiSE*, volume 3520 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2005.

[90] D. Sabin and E.C. Freuder. Configuration as composite constraint satisfaction. *Artificial Intelligence and Manufacturing Research Planning Workshop*, pages 153–161, 1996.

[91] Daniel Sabin and Rainer Weigel. Product configuration frameworks - a survey. *IEEE Intelligent Systems*, 13(4):42–49, 1998.

[92] Meinolf Sellman and Pascal Van Hentenrick. Structural symmetry breaking. In *proceedings of IJCAI 2005,*. Morgan Kaufman, 2005.

[93] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artif. Intell.*, 138(1-2):181–234, 2002.

[94] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.

[95] Evren Sirin, James A. Hendler, and Bijan Parsia. Semi-automatic composition of web services using semantic descriptions. In Jean Bézivin, Jiankun Hu, and Zahir Tari, editors, *WSMAI*, pages 17–24. ICEIS Press, 2003.

[96] Evren Sirin, Bijan Parsia, Dan Wu, James A. Hendler, and Dana S. Nau. Htn planning for web service composition using shop2. *J. Web Sem.*, 1(4):377–396, 2004.

[97] Timo Soininen, Ilkka Niemelä, Juha Tiihonen, and Reijo Sulonen. Representing configuration knowledge with weight constraint rules. In *Answer Set Programming*, 2001.

[98] Christine Solnon. Solving permutation constraint satisfaction problems with artificial ants. In Werner Horn, editor, *ECAI*, pages 118–122. IOS Press, 2000.

[99] Christine Solnon. Ants can solve constraint satisfaction problems. *IEEE Trans. Evolutionary Computation*, 6(4):347–357, 2002.

[100] Christine Solnon. Boosting aco with a preprocessing step. In Stefano Cagnoni, Jens Gottlieb, Emma Hart, Martin Middendorf, and Günther R. Raidl, editors, *EvoWorkshops*, volume 2279 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2002.

[101] J. M. Spivey. *The Z Notation: a reference manual*. Prentice Hall originally, now J.M. Spivey, 2001.

[102] Harald Storrle and Jan Hendrik Hausmann. Towards a formal semantics of uml 2.0 activities. 2004.

[103] Markus Stumptner. An overview of knowledge-based configuration. *AI Communications*, 10(2):111–125, June 1997.

[104] Markus Stumptner and Alois Haselböck. A generative constraint formalism for configuration problems. In Pietro Torasso, editor, *AI*IA*, volume 728 of *Lecture Notes in Computer Science*, pages 302–313. Springer, 1993.

[105] Thomas Stützle and Holger H. Hoos. Max-min ant system. *Future Generation Comp. Syst.*, 16(8):889–914, 2000.

[106] Paolo Traverso and Marco Pistore. Automated composition of semantic web services into executable processes. In Sheila A. McIlraith, Dimitris Plexousakis, and Frank van Harmelen, editors, *International Semantic Web Conference*, volume 3298 of *Lecture Notes in Computer Science*, pages 380–394. Springer, 2004.

[107] Wil van der Aalst, Marlon Dumas, C. Ouyang, Anne Rozinat, and H. M. W. Verbeek. Choreography conformance checking: An approach based on bpel and petri nets. In Frank Leymann, Wolfgang Reisig, Satish R. Thatte, and Wil van der Aalst, editors, *The Role of Business Processes in Service Oriented Architectures*, number 06291 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[108] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. Yawl: yet another workflow language. *Inf. Syst.*, 30(4):245–275, 2005.

[109] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[110] Mirko Viroli. Towards a formal foundation to orchestration languages. *Electr. Notes Theor. Comput. Sci.*, 105:51–71, 2004.

[111] Valdis Vitolins and Audris Kalnins. Semantics of uml 2.0 activity diagram for business modeling by means of virtual machine. In *EDOC*, pages 181–194. IEEE Computer Society, 2005.

[112] Rainer Weigel and Boi Faltings. Compiling constraint satisfaction problems. *Artif. Intell.*, 115(2):257–287, 1999.

[113] Petia Wohed, Wil M. P. van der Aalst, Marlon Dumas, Arthur H. M. ter Hofstede, and Nick Russell. Pattern-based analysis of the control-flow perspective of uml activity diagrams. In Delcambre et al. [24], pages 63–78.

[114] Zixin Wu, Ajith Ranabahu1, Karthik Gomadam, Amit P. Sheth, and John A. Miller. Automatic composition of semantic web services using process and data mediation. Technical report, 2007.

Abstract
This thesis aims at using constraint programming, and more precisely finite model search for constraint object models, to achieve symbolic reasoning and address artificial intelligence problems.
A the denotational level, we illustrate the expressive power of the chosen formalism through the description of a theoretical and experimental framework addressing a very modern challenge: the automatic composition of semantic web services. This framework, developped during the DIP European project and prototyped using ILOG's tool JConfigurator, has been integrated in the project's architecture and tested on industrial use cases.
At the operational level, we describe algorithms dealing with the inherent combinatorial explosion of enumerative methods. We propose a pseudo-linear time algorithm that approximates colored directed acyclic graphs canonicity detection, allowing early backtrack when generating isomorphic configurations during the search. The theoretical results are backed by a range of experiments. We also propose a stochastic method based on simulated ant colony behaviour which handles original first-order logic issues. There again we present experimental results.

Keywords: constrained object models, finite model search, configuration, constraint programming, semantic web services, composition, graph isomorphism, ant colony optimization.

Résumé
Cette thèse a pour but d'utiliser la programmation par contraintes, et plus précisément la recherche de modèles finis pour les modèles objets contraints, dans le raisonnement symbolique et la résolution de problèmes d'intelligence artificielle.
Au niveau dénotationnel, nous illustrons la puissance expressive du formalisme choisi à travers la description d'un cadre théorique et expérimental pour un challenge moderne: la composition automatique de services web sémantiques. Ce cadre, développé durant le projet Européen DIP et prototypé à l'aide de l'outil JConfigurator d'ILOG, a été intégré au sein de l'architecture du projet et testé sur des scénarios industriels.
Au niveau opérationnel, nous décrivons des algorithmes traitant l'explosion combinatoire inhérente aux méthodes énumératives. Nous proposons un algorithme pseudo-linéaire en temps pour la détection de canonicité de graphes orientés acycliques colorés, permettant, au cours de la recherche, de "backtracker" sur des configurations isomorphes. Les résultats théoriques sont appuyés par une série d'expérimentations. Nous proposons également une méthode stochastique, basée sur le comportement simulé d'une colonie de fourmi, qui traite des problèmes originaux posés par la logique du premier-ordre. De nouveau nous présentons des résultats expérimentaux.

Mots-clés: modèles objets contraints, recherche de modèles finis, configuration, programmation par contraintes, services web sémantiques, composition, isomorphisme de graphe, optimisation par colonie de fourmis.

Laboratoire des Sciences de l'Information
et des Systèmes (LSIS) - UMR CNRS 6168

ILOG S.A