



HAL
open science

A Guarded Attribute Grammar Based Model for User Centered, Distributed, and Collaborative Case Management Case of the Disease Surveillance Process

Robert Nsaibirni

► **To cite this version:**

Robert Nsaibirni. A Guarded Attribute Grammar Based Model for User Centered, Distributed, and Collaborative Case Management Case of the Disease Surveillance Process. Computer Science [cs]. Université de Yaoundé I, 2019. English. NNT : . tel-02263094

HAL Id: tel-02263094

<https://inria.hal.science/tel-02263094v1>

Submitted on 2 Aug 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

University of Yaounde I
STG - MIBA Doctoral Research Unit
Faculty of Science, Department of Computer Science

A Guarded Attribute Grammar Based Model for User Centered, Distributed, and Collaborative Case Management

Case of the Disease Surveillance Process

NSAIBIRNI, Robert Fondze Jr

Thesis Directors: Eric BADOUEL, Georges-Edouard KOUAMOU

Thesis Supervisor: Maurice TCHUENTE

Co-Supervisor: Gaëtan TEXIER

Defended in partial fulfillment of the requirements for the degree of
Doctor of Philosophy (Ph.D) in Computer Science of the University of Yaounde I.

Date: April 24, 2019

Before the Jury made up of:

FOUDA NDJODO Marcel

Professeur, Université de Yaoundé I, President

TCHUENTE Maurice

Professeur, Université de Yaoundé I, Rapporteur

KOUAMOU Georges-Edouard

Maître de Conférences, Université de Yaoundé I, Rapporteur

BADOUEL Eric

Directeur de Recherche, INRIA Rennes (France), Rapporteur

TEXIER Gaëtan

MD/PhD, Université Aix-Marseille (France), Rapporteur

AGOSTINI Alessandra

Professeur, Université de Milano-Bicocca (Italie), Membre

ATSA ETOUNDI Roger

Professeur, Université de Yaoundé I, Membre

NDOUNDAM René

Maître de Conférences, Université de Yaoundé I, Membre

Abstract

Dynamic processes in which users need to work together and collaborate in myriad ways on process models defined on-the-fly are fast becoming the rule rather than the exception. This thesis presents the design of a purely declarative modelling approach for dynamic, collaborative, user-centered, and data-driven processes. First, we organize the work of a user into task hierarchies which we model as mindmaps, which are trees used to visualize, organize, and log information about tasks in which the user is involved. We introduce the model of *guarded attribute grammar*, or GAG, to help the automation of updating such maps. A GAG consists of an underlying grammar, that specifies the logical structure of the map, with semantic rules which are used both to govern the evolution of the tree structure (how an open node may be refined to a sub-tree) and to compute the values of some of its attributes. The map enriched with this extra information and with high level constructs for task dependencies; collaboration and user-interactions is termed an *active workspace* or AW. Communication between AWs is essentially through exchange of messages without a shared memory thus enabling convenient distribution on an asynchronous architecture. Lastly, we introduce a language syntax for GAG specification and design a prototype that includes an internal domain specific language (in Haskell) for their specification and a graphical user interface to simulate its execution in a distributed environment. We motivate our approach and illustrate its language syntax and features on a case study for a disease surveillance system.

Keywords:

Business Process Management, Case Management, Dynamic Processes, Collaborative Systems, Business Artifacts, Guarded Attribute Grammars, Active-Workspaces, Disease Surveillance Process.

Résumé

De plus en plus, les utilisateurs collaborent de multiples façons sur des processus dynamiques construits de manière progressive. Dans cette thèse, nous concevons une nouvelle approche déclarative de modélisation des processus dynamiques, centrés sur l'utilisateur et dirigés par les données. Tout d'abord, nous organisons le travail d'un utilisateur par des hiérarchies des tâches, représentées par des cartes heuristiques (arbre de tâches). Ces derniers sont utilisés pour visualiser, organiser, et sauvegarder les informations sur les tâches menés par l'utilisateur. Nous introduisons ensuite le modèle des *grammaires attribuées gardées*, ou GAG, pour faciliter l'automatisation de la manipulation de telles cartes. Une GAG consiste en une grammaire sous-jacente, qui spécifie la structure logique de la carte, avec des règles sémantiques qui servent à la fois à gouverner l'évolution de l'arbre des tâches (raffinement des nœuds ouverts) et à calculer les valeurs de certains de ses attributs. La carte enrichie de ces informations supplémentaires et d'autres concepts de haut niveau pour les dépendances entre les tâches, la collaboration et les interactions utilisateur est appelée *Active Workspace* ou AW. La communication entre AWs est essentiellement par échange de messages permettant ainsi une implémentation commode sur une architecture distribuée et asynchrone. Enfin, nous décrivons une syntaxe de langage pour la spécification des processus en utilisant les GAGs et concevons un prototype qui inclut un langage spécifique au domaine, interne à Haskell, pour leur spécification et une interface utilisateur graphique pour la simulation de l'exécution dans un environnement distribué. Nous motivons notre approche et illustrons sa syntaxe et ses caractéristiques sur une étude de cas portant sur le processus de surveillance épidémiologique.

Mots clés :

Business Process Management, Case Management, Processus Dynamiques, Systèmes Collaboratifs, Business Artifacts, Grammaires Attribuées Gardées, Active-Workspaces, Surveillance Epidémiologiques.

A Guarded Attribute Grammar based-model for User-Centered, Distributed and Collaborative Case Management: Case of the Disease Surveillance Process

Nsaibirni Robert Fondze Jr

Extended Abstract

1 Thesis Objective

This work focuses on distributed, user-centric collaborative process models that provide the flexibility and adaptability needed for case management systems, such as early warning systems for disease outbreak detection and control. These systems involve many geographically distant stakeholders, playing different roles on tasks that only become known when data becomes available, and working mostly in limited connectivity settings. Our objective is to provide each of these users with a workspace that shows at each instant, the current tasks that requires the user's intervention based on the available data, the information attached to these tasks and a list of actions the user can perform to resolve these tasks. Also, the system should be flexible enough to allow for data-centric enactment, easy runtime reconfiguration, on-the-fly design and effective collaboration. These strong constraints led to an interest in a declarative model in the form of rules: the model of attribute grammars.

2 Work Performed

The research context is Case Management, a sub-domain of Business Process Modelling (BPM). Specifically, the work focuses on dynamic, user-centric, and data-driven processes. Dynamic processes are those in which (i) data plays a key role in the selection and scheduling of work and (ii) stakeholders interact (collaborate) in performing a process that is only progressively specified.

Two illustrative examples were explored to show on real world cases extracted from scenarios of epidemiological surveillance, the relevance of having a flexible and expressive model allowing stakeholders to (i) control the execution of the process (ii) perform on-the-fly process modelling (iii) collaborate with

other stakeholders and (iv) easily and collectively make decisions.

This context is complemented by a literature review of tools and methods for process modelling in general and dynamic processes in particular, and a presentation of two key concepts of the work: collaboration & user interactions, and process flexibility. The literature review is completed with a presentation of the disease surveillance process and its properties that characterize it as being data-driven, user-centred, and collaborative.

2.1 Proposed Model

The proposed model, called the Guarded Attribute Grammar (GAG) model, assumes the use of the hierarchical task decomposition technique for the analysis and identification of the tasks to be performed to solve a problem. This decomposition is modelled by rewriting rules in the form of productions of an attribute grammar. Attributes of the grammar are used in semantic rules to model the flow of information between a task and its subtasks and to introduce constraints on the data, which will be useful later to drive the execution of the process.

The properties of the model are based on three pillars: composition, distribution, and a form of termination (soundness). Composition adds modularity to GAG based modelling and hence support for iterative modelling, Distribution guarantees deployment on a distributed and asynchronous architecture, and soundness ensures that all well-formed GAG specifications terminate. Soundness is undecidable in general cases but can be easily verified on a particular family of GAGs.

The concept Active Workspaces (AW), describes the layer above the GAG model that contains the actions available to a particular user (the owner of the AW) and provides the tools needed to (i) carry out these actions in conformity with the GAG operational semantics and (ii) to interact with the system and with other active workspaces. A user's AW contains the GAGs of the services offered by the user as well as the artifacts (execution trees) that he initiated.

These results were published in ACM's ACR journal in 2015, and a demonstration of its applicability to disease surveillance was presented at the CRI'15 conference in Yaounde. Also, a paper presenting the requirements for the modelling of a dynamic process such as the disease surveillance process was presented at the HIMS'2016 conference in Las Vegas, and another publication focusing on collaboration tools with GAGs was presented at the CARI in 2016 in Hammamet.

2.2 Experimentation and validation

The Active Workspaces model was enriched with formal tools to facilitate the modelling of services and roles, temporal dependencies between tasks (sequence, parallelism, optionality, loops, dialog, and checklists), interactions between users (single-/multi-user service calls), time constraints, and the use of semantic functions and Boolean formulae to enrich the guards of a GAG specification.

These tools were implemented in a prototype of the model that contains: a Haskell-internal Domain Specific Language (GAG-DSL) and a runtime engine with a graphical interface that simulates execution in a distributed environment. An excerpt of the disease surveillance process for human influenza was completely specified using the GAG-DSL and its execution simulated using the prototype.

The GAG-DSL can be updated on-the-fly and the updates are built and available for subsequent executions. Also, the prototype GUI demonstrates the data drivenness of process enactment with data objects built progressively in push mode, and gives users ample control on how the process unfolds. The user decides which actions are performed on pending tasks, who receives service calls and in what form (single/multi-user), and how values for certain inherited attributes are provided. Finally, the artifacts give users great visibility over process enactment and orchestration.

3 Conclusion

The work presented covers a broad spectrum of works combining (1) theoretical work of formalization of a model (AW / GAG) including the formal study of its properties, (2) the elaboration of a methodology for the specification of a system using such a model, (3) the development of a demonstrator and finally (3) a demonstration of the applicability of the model through the specification of scenarios from disease surveillance.

Our objective in this thesis was to design a suitable (adaptive case management) model for distributed collaborative systems such as the disease surveillance system. Using examples and literature from disease surveillance systems, we characterized such systems as requiring high levels of flexibility at both design and run-time. The systems are artifact-centric, user-driven, and collaborative, and their modelling should include all of the following aspects: (i) iterative (incremental) design, (ii) collaboration and user interactions, (iii) techniques to leverage uncertainty and exceptions, (iv) decision-making support. We proposed a model that naturally supports iterative modelling, concurrent and flexible process enactment, collaboration and user interactions in a distributed environment.

Un Modèle basé sur les Grammaires Attribuées Gardées pour les Processus Dynamiques, Centrés sur l'Utilisateur, Distribués et Collaboratifs: Cas de la Surveillance Epidémiologique

Nsaibirni Robert Fondze Jr

Résumé Etendu

1 Problématique de la Thèse

Les travaux menés portent sur un modèle de travail collaboratif distribué, centré sur les utilisateurs offrant la souplesse et l'adaptabilité nécessaires aux systèmes de gestion de crises, et en particulier les systèmes de détection précoce des épidémies. Ces systèmes mettent en jeu un grand nombre d'intervenants, jouant des rôles différents sur des tâches qui ne sont connues qu'après la production des données, et pouvant se trouver géographiquement distants. Chaque intervenant doit pouvoir travailler aussi bien en mode connecté que déconnecté. Notre objectif est de fournir à chaque utilisateur, un espace de travail lui montrant à chaque instant les actions en cours qui le concernent, les informations qui y sont attachées et les actions qu'il peut choisir afin de faire progresser ces tâches. Par ailleurs le système doit pouvoir être facilement reconfigurable pour permettre aux utilisateurs de changer dynamiquement la façon dont ils souhaitent opérer pour exécuter les tâches qui leur sont confiées. Ces contraintes fortes ont conduit à s'intéresser à un modèle déclaratif sous forme de règles : le modèle des grammaires attribuées.

2 Travaux Effectués

Le contexte de recherche est le Case Management, un sous-domaine du Business Process Modelling (BPM). Plus particulièrement, les travaux portent sur les outils de modélisation des processus dynamiques, centrés sur les utilisateurs, et dirigés par les données. Les processus dynamiques sont ceux dans lesquels (i) les données jouent un rôle clé dans le choix et l'ordonnancement des travaux et (ii) les intervenants interagissent (collaborent) dans l'exécution d'un processus qui n'est que progressivement spécifié.

Deux exemples illustratifs ont été explorés pour montrer sur des cas réels extraits des scénarios de la surveillance épidémiologique, la pertinence d’avoir un modèle flexible et expressif permettant aux intervenants de (i) contrôler l’exécution du processus, (ii) définir et planifier l’exécution des nouvelles tâches pendant l’exécution, (iii) collaborer avec d’autres intervenants, et (iv) faciliter la prise de décision (collective).

Ce contexte est complété par une revue de la littérature des outils et méthodes de modélisation des processus en général et des processus dynamiques en particulier et une présentation de deux concepts clés du travail : la collaboration et les interactions utilisateurs, et la flexibilité des processus. Cet état des l’art est terminé par une présentation de ce qu’est la surveillance épidémiologique et de ses propriétés qui permettent de la caractériser comme étant dirigée par les données, centrée sur l’utilisateur, et collaborative.

2.1 Modèle Proposé

Le modèle proposé est basé sur l’approche des Grammaires Attribuées Gardées (GAG) qui suppose l’utilisation de la technique de décomposition hiérarchique pour l’analyse et l’identification des tâches à exécuter pour résoudre un problème. Cette décomposition est modélisée par des règles de réécriture sous forme de productions d’une grammaire attribuée. Les attributs de la grammaire sont utilisés dans les règles sémantiques pour modéliser le flot d’information entre une tâche et ses sous-tâches et pour introduire les contraintes sur les données, qui seront utiles par la suite pour diriger l’exécution du processus.

Les propriétés du modèle reposent sur trois piliers : la composition, la distribution, et une forme de terminaison (soundness). La composition permet de rendre modulaire la spécification des processus avec les GAG, la distribution garanti le déploiement sur une architecture distribuée et asynchrone, et le soundness assure que toute modélisation est bien formée et termine. Ce dernier est indécidable dans les cas général mais peut être facilement vérifiée sur des familles particulières des GAG.

Le concept des espaces de travail actifs (Active Workspaces -AW) qui sont la couche au-dessus des GAGs rassemble les actions d’un utilisateur et fournit des outils lui permettant d’interagir avec le système et avec d’autres espaces de travail actifs. L’AW d’un utilisateur contient les GAGs des services qu’il offre ce dernier ainsi que les artefacts (arbres d’exécution) qu’il a initiés.

Ces résultats ont été publiés au journal ACR de l’ACM en 2015, et une démonstration de son applicabilité sur un système réel qu’est la surveillance épidémiologique a été présentée à la conférence CRI’15 à Yaoundé. Ensuite, une étude sur les besoins que doivent fournir un outil de modélisation de la surveillance épidémiologique a été présentée à la conférence HIMS’2016 à Las Vegas, et une autre publication mettant l’accent sur les outils de collaboration avec les GAGs a été présentée au CARI en 2016 à Hammamet.

2.2 Expérimentation et Validation

Pour valider le modèle, nous avons conçu au-dessus du modèle de base, les outils formels permettant de modéliser les services et les rôles, les dépendances temporelles entre les tâches (séquence, parallélisme, optionalité, les boucles, le dialogue, et les checklists), les interactions entre utilisateurs, les contraintes temporelles, et l'utilisation des fonctions sémantiques et formules booléennes pour enrichir la spécification des GAGs.

Ces outils sont implémentés dans un prototype du modèle qui contient : un langage dédié interne à Haskell (GAG-DSL) et un moteur d'exécution avec une interface graphique qui permet de simuler l'exécution dans un environnement. Un extrait du système de surveillance syndromique de la grippe humaine est modélisé dans le GAG-DSL et son exécution simulée.

Une spécification de processus dans le GAG-DSL peut être modifiée à chaud et ces modifications sont compilées et utilisables dans les prochaines exécutions. Le prototype démontre aussi l'exécution dirigée par les données et le contrôle dont dispose l'utilisateur sur cette exécution. L'utilisateur décide des actions à mener, des destinataires à qui envoyer des appels de services, et comment certaines données en entrées sont fournies. Enfin, les artefacts donnent aux intervenants une bonne visibilité sur le déroulement du processus.

3 Conclusion

Le travail présenté couvre un large spectre de travaux combinant (1) un travail théorique de formalisation d'un modèle (AW/GAG) incluant l'étude formelle de ses propriétés, (2) l'élaboration d'une méthodologie pour la spécification d'un système à l'aide d'un tel modèle, (3) le développement d'un démonstrateur et enfin (3) une étude du problème de détection précoce des épidémies, qui avait été choisi comme cas d'étude, afin d'aboutir à une compréhension fine de cette problématique et d'en apporter des solutions.

Notre objectif dans cette thèse était de concevoir un modèle (de gestion adaptative des cas) approprié pour les systèmes collaboratifs distribués tels que le système de surveillance des maladies. En utilisant des exemples et de la documentation provenant des systèmes de surveillance des maladies, nous avons caractérisé ces systèmes comme exigeant des niveaux élevés de flexibilité à la fois au niveau de la conception et de l'exécution. Ces systèmes sont artefact-centrique, centrés sur l'utilisateur et collaboratifs, et leur modélisation doit inclure tous les aspects suivants : (i) conception itérative (incrémentielle), (ii) collaboration et interactions avec les utilisateurs, (iii) techniques pour lever l'incertitude et (iv) l'aide à la décision. Nous avons proposé un modèle qui prend naturellement en charge la modélisation itérative, l'exécution concurrente et flexible, la collaboration et les interactions des utilisateurs dans un environnement distribué.

Acknowledgements

I would like to express gratitude and thanks to my supervisors for their confidence, commitment, and unflinching guidance throughout my PhD years.

I also specially thank:

- The following research laboratories:
 - UMMISCO (Unité Mixte Internationale de Modélisation Mathématique et Informatiques des Systèmes Complexes) - IRD UMI 209
 - LIRIMA (Laboratoire International de Recherche en Informatique et en Mathématiques Appliquées)
 - INRIA (Institut National de Recherche en Informatique et en Automatique) Rennes SUMO (SUPERVISION of large MODular and distributed systems) team.
- The Centre Pasteur of Cameroon (CPC), most especially the Public-Health/Epidemiology and Virology services.
- The lecturers of the Department of Computer Science of the University of Yaounde I
- My colleagues at the Cameroon ANRS Site and at the Centre Pasteur of Cameroon.
- My fellow mates
- My family and friends

For their availability, support, and advice throughout the years. I will forever be grateful.


This work was carried out within the framework of a collaboration initiated by UMMISCO between the University of Yaounde I (UY1) and CPC, with EPICAM (a collaborative project with UY1-CPC-PNLT-MEDES) as a major output. Indeed the idea to pursue research work on an appropriate disease surveillance architecture for limited resource settings (Cameroon for example), emanated from this collaboration.

This work was supported by the U.S. Department of Health and Human Services (DHHS) via the International Network of Pasteur Institutes and the ASIDE Project (Alerting and Surveillance for Infectious Disease Epidemics) to the Centre Pasteur of Cameroon.

Dedication

To the NSAIBIRNIs

Dad (Kitiyfo), Mum (Mami Yaa), Miranda, Lee, Gael, and Drew.

| | | |
|--|---|---|
| UNIVERSITÉ OF YAOUNDÉ I Faculté des Sciences Division de la Programmation et du Suivi des Activités Académiques |  | THE UNIVERSITY OF YAOUNDE I Faculty of Science Division of Programming and Follow-up of Academic Affairs |
| LISTE DES ENSEIGNANTS PERMANENTS | | LIST OF PERMANENT TEACHING STAFF |

ACADEMIC YEAR 2017/2018

(By Department and Grade)

REVISION DATE : 10 Mars 2018

ADMINISTRATION

DEAN : AWONO ONANA Charles, *Professor*

VICE DEAN / DPSAA: DONGO Etienne, *Professor*

VICE DEAN / DSSE: OBEN Julius ENYONG, *Professor*

VICE DEAN / DRC: MBAZE MEVA'A Luc Léonard, *Associate Professor*

Chief of the Administration and Finance Division: NDOYE FOE Marie C. F., *Associate Professor*

Chief of the Academic Affairs, Student followup, and Research Division: ABOSSOLO Monique, *Associate Professor*

| 1- DEPARTMENT OF BIOCHEMISTRY (BC) (41) | | | |
|--|------------------------------------|---------------------|---------------------------|
| N° | NAME & SURNAME | GRADE | OBSERVATIONS |
| 1 | FEKAM BOYOM Fabrice | Professor | In Service |
| 2 | MBACHAM FON Wilfried | Professor | In Service |
| 3 | MOUNDIPA FEWOU Paul | Professor | Head of Department |
| 4 | NINTCHOM PENLAP V. épouse BENG | Professor | In Service |
| 5 | OBEN Julius ENYONG | Professor | In Service |
| 6 | ATOGHO Barbara Mma | Associate Professor | In Service |
| 7 | BELINGA née NDOYE FOE M. C. F. | Associate Professor | Chef DAF / FS |
| 8 | BIGOGA DIAGA Jude | Associate Professor | In Service |
| 9 | BOUDJEKO Thaddée | Associate Professor | In Service |
| 10 | EFFA NNOMO Pierre | Associate Professor | In Service |
| 11 | FOKOU Elie | Associate Professor | In Service |
| 12 | KANSCI Germain | Associate Professor | In Service |
| 13 | NANA Louise épouse WAKAM | Associate Professor | In Service |
| 14 | NGONDI Judith Laure | Associate Professor | In Service |
| 15 | NGUEFACK Julienne | Associate Professor | In Service |
| 16 | NJAYOU Frédéric Nico | Associate Professor | In Service |
| 17 | ACHU Merci BIH | Senior Lecturer | In Service |
| 18 | BIYITI BI ESSAM née AKAM ADA L. | Senior Lecturer | CT MINRESI |
| 19 | DEMMANO Gustave | Senior Lecturer | In Service |
| 20 | DJOKAM TAMO Rosine | Senior Lecturer | In Service |
| 21 | DJUIDJE NGOUNOUE Marcelline | Senior Lecturer | In Service |
| 22 | DJUUKWO NKONGA Ruth Viviane | Senior Lecturer | In Service |
| 23 | EVEHE BEBANDOUE Marie- Solange | Senior Lecturer | In Service |
| 24 | EWANE Cécile Anne | Senior Lecturer | In Service |
| 25 | KOTUE KAPTUE Charles | Senior Lecturer | In Service |

| | | | |
|----|------------------------------------|--------------------------------|----------------|
| 26 | LUNGA Paul KEILAH | Senior Lecturer | In Service |
| 27 | MBONG ANGIE M. Mary Anne | Senior Lecturer | In Service |
| 28 | MOFOR née TEUGWA Clotilde | Senior Lecturer | CE SEP MINESUP |
| 29 | NJAYOU Frédéric Nico | Senior Lecturer | In Service |
| 30 | Palmer MASUMBE NETONGO | Senior Lecturer | In Service |
| 31 | TCHANA KOUATCHOUA Angèle | Senior Lecturer | In Service |
| 32 | PACHANGOU NSANGOU Sylvain | Senior Lecturer | In Service |
| 33 | DONGMO LEKAGNE Joseph Blaise | Senior Lecturer | In Service |
| 34 | FONKOUA Martin | Senior Lecturer | In Service |
| 35 | BEBOY EDZENGUELE Sara Nathalie | Senior Lecturer | In Service |
| 36 | DAKOLE DABOY Charles | Senior Lecturer | In Service |
| 37 | MANANGA Marlyse Joséphine | Senior Lecturer | In Service |
| 38 | MBOUCHE FANMOE Marceline Joëlle | Assistant Lecturer Lecturer | In Service |
| 39 | BEBEE Fadimatou | Assistant Lecturer Lecturer | In Service |
| 40 | TIENTCHEU DJOKAM Leopold | Assistant Lecturer | In Service |

2- DEPARTMENT OF ANIMAL BIOLOGY AND PHYSIOLOGY (BPA) (44)

| | | | |
|----|------------------------------|---------------------|-------------------------------|
| 1 | BILONG BILONG Charles-Félix | Professor | Head of Department |
| 2 | DIMO Théophile | Professor | In Service |
| 3 | DJIETO LORDON Champlain | Professor | In Service |
| 4 | ESSOMBA née NTSAMA MBALA | Professor | <i>VDean/FMSB/UYY</i> |
| 5 | FOMENA Abraham | Professor | In Service |
| 6 | KAMTCHOING Pierre | Professor | IN SERVICE |
| 7 | NJAMEN Dieudonné | Professor | In Service |
| 8 | NJIOKOU Flobert | Professor | In Service |
| 9 | NOLA Moïse | Professor | In Service |
| 10 | TAN Paul VERNYUY | Professor | In Service |
| 11 | TCHUEM TCHUENTE Louis Albert | Professor | <i>Coord. Progr. MINSANTE</i> |
| 12 | AJEAGAH Gidéon AGHAINDUM | Associate Professor | Chief of Service DPER |
| 13 | DZEUFLET DJOMENI Paul Désiré | Associate Professor | In Service |
| 14 | FOTO MENBOHAN Samuel | Associate Professor | In Service |
| 15 | KAMGANG René | Associate Professor | <i>C.S. MINRESI</i> |
| 16 | KEKEUNOU Sévilor | Associate Professor | In Service |
| 17 | MEGNEKOU Rosette | Associate Professor | In Service |
| 18 | MONY Ruth épouse NTONE | Associate Professor | In Service |
| 19 | TOMBI Jeannette | Associate Professor | In Service |
| 20 | ZEBAZE TOGOUET Serge Hubert | Associate Professor | In Service |
| 21 | ALENE Désirée Chantal | Senior Lecturer | In Service |
| 22 | ATSAMO Albert Donatien | Senior Lecturer | In Service |
| 23 | BELLET EDIMO Oscar Roger | Senior Lecturer | In Service |
| 24 | BILANDA Danielle Claude | Senior Lecturer | In Service |
| 25 | DJIOGUE Séfirin | Senior Lecturer | In Service |
| 26 | DONFACK Mireille | Senior Lecturer | In Service |
| 27 | GOUNOUE KAMKUMO Raceline | Senior Lecturer | In Service |
| 28 | LEKEUFACK FOLEFACK Guy B. | Senior Lecturer | In Service |
| 29 | MAHOB Raymond Joseph | Senior Lecturer | In Service |
| 30 | MBENOUN MASSE Paul Serge | Senior Lecturer | In Service |

| | | | |
|----|---------------------------------|--------------------|---------------|
| 31 | MOUNGANG Luciane Marlyse | Senior Lecturer | In Service |
| 32 | MVEYO NDANKEU Yves Patrick | Senior Lecturer | In Service |
| 33 | NGOUATEU KENFACK Omer Bébé | Senior Lecturer | In Service |
| 34 | NGUEGUIM TSOFAK Florence | Senior Lecturer | In Service |
| 35 | NGUEMBOK | Senior Lecturer | In Service |
| 36 | NJATSA Hermine épouse MEGAPTCHE | Senior Lecturer | In Service |
| 37 | NJUA Clarisse Yafi | Senior Lecturer | CD/UBa |
| 38 | NOAH EWOTI Olive Vivien | Senior Lecturer | In Service |
| 39 | TADU Zephyrin | Senior Lecturer | In Service |
| 40 | YEDE | Senior Lecturer | In Service |
| 41 | ETEME ENAMA Serge | Assistant Lecturer | In Service |
| 42 | KANDEDA KAVAYE Antoine | Assistant Lecturer | In Service |
| 43 | KOGA MANG DOBARA | Assistant Lecturer | In Service |

3- DEPARTMENT OF PLANT BIOLOGY AND PHYSIOLOGY (BPV) (26)

| | | | |
|----|------------------------------------|---------------------|--------------------------------|
| 1 | AMBANG Zachée | Professor | Chef Division/UYII |
| 2 | BELL Joseph Martin | Professor | In Service |
| 3 | YOUMBI Emmanuel | Professor | Head of Department |
| 4 | MOSSEBO Dominique Claude | Professor | In Service |
| 5 | BIYE Elvire Hortense | Associate Professor | In Service |
| 6 | DJOCGOUE Pierre François | Associate Professor | In Service |
| 7 | KENGNE NOUMSI Ives Magloire | Associate Professor | In Service |
| 8 | MALA Armand William | Associate Professor | In Service |
| 9 | NDONGO BEKOLO | Associate Professor | <i>CE / MINRESI</i> |
| 10 | NGONKEU MAGAPTCHE Eddy L. | Associate Professor | In Service |
| 11 | ZAPFACK Louis | Associate Professor | In Service |
| 12 | MBARGA BINDZI Marie Alain | Associate Professor | CT/Univ Dschang |
| 13 | MBOLO Marie | Associate Professor | In Service |
| 14 | ANGONI Hyacinthe | Senior Lecturer | In Service |
| 15 | MAHBOU SOMO TOUKAM. Gabriel | Senior Lecturer | In Service |
| 16 | ONANA JEAN MICHEL | Senior Lecturer | In Service |
| 17 | GOMANDJE Christelle | Senior Lecturer | In Service |
| 18 | NGODO MELINGUI Jean Baptiste | Senior Lecturer | In Service |
| 19 | NGALLE Hermine BILLE | Senior Lecturer | In Service |
| 20 | NGOUO Lucas Vincent | Senior Lecturer | In Service |
| 21 | NSOM ZAMO Annie Claude épouse PIAL | Senior Lecturer | <i>Expert national /UNESCO</i> |
| 22 | TONFACK Libert Brice | Senior Lecturer | In Service |
| 23 | TSOATA Esaïe | Senior Lecturer | In Service |
| 24 | DJEUANI Astride Carole | Assistant Lecturer | In Service |
| 25 | MAFFO MAFFO Nicole Liliane | Assistant Lecturer | In Service |
| 26 | NNANGA MEBENGA Ruth Laure | Assistant Lecturer | In Service |
| 27 | NOUKEU KOUAKAM Armelle | Assistant Lecturer | In Service |

4- DEPARTMENT OF INORGANIC CHEMISTRY (CI) (33)

| | | | |
|---|---------------------------------|-----------|----------------------------------|
| 1 | AGWARA ONDOH Moïse | Professor | <i>Vice Rector Univ ,Bamenda</i> |
| 2 | ELIMBI Antoine | Professor | In Service |
| 3 | Florence UFI CHINJE épouse MELO | Professor | <i>Rector Univ.Ngaoundere</i> |
| 4 | GHOGOMU Paul MINGO | Professor | <i>Director of Cabinet PM</i> |
| 5 | LAMINSI Samuel | Professor | In Service |
| 6 | NANSEU Charles Péguy | Professor | In Service |

| | | | |
|----|--------------------------------|---------------------|---------------------------------------|
| 7 | NDIFON Peter TEKE | Professor | <i>ISI MINRESI/Head of Department</i> |
| 8 | NENWA Justin | Professor | In Service |
| 9 | NGAMENI Emmanuel | Professor | <i>DEAN FS Univ. Dschang</i> |
| 10 | BABALE née DJAM DOUDOU | Associate Professor | <i>Chargée Mission P.R.</i> |
| 11 | DJOUFAC WOU MFO Emmanuel | Associate Professor | In Service |
| 12 | KEMEGNE MBOUGUEM Jean C. | Associate Professor | In Service |
| 13 | KONG SAKEO | Associate Professor | <i>Chargé de Mission au P. M.</i> |
| 14 | NDIKONTAR Maurice KOR | Associate Professor | <i>VICE DEAN Univ. Bamenda</i> |
| 15 | NGOMO Horace MANGA | Associate Professor | <i>VC/UB</i> |
| 16 | NJIOMOU C. épouse DJANGANG | Associate Professor | In Service |
| 17 | YOUNANG Elie | Associate Professor | In Service |
| 18 | ACAYANKA Elie | Senior Lecturer | In Service |
| 19 | EMADACK Alphonse | Senior Lecturer | In Service |
| 20 | KAMGANG YOUNBI Georges | Senior Lecturer | In Service |
| 21 | NDI NSAMI Julius | Senior Lecturer | In Service |
| 22 | NJOYA Dayirou | Senior Lecturer | In Service |
| 23 | PABOUDAM GBAMBIE A. | Senior Lecturer | In Service |
| 24 | TCHAKOUTE KOUAMO Hervé | Senior Lecturer | In Service |
| 25 | BELIBI BELIBI Placide Désiré | Senior Lecturer | In Service |
| 26 | CHEUMANI YONA Arnaud M. | Senior Lecturer | In Service |
| 27 | NYAMEN Linda Dyorisse | Senior Lecturer | In Service |
| 28 | KENNE DEDZO GUSTAVE | Senior Lecturer | In Service |
| 29 | KOUOTOU DAOUDA | Senior Lecturer | In Service |
| 30 | MAKON Thomas Beauregard | Senior Lecturer | In Service |
| 31 | MBEY Jean Aime | Senior Lecturer | In Service |
| 32 | NCHIMI NONO KATIA | Senior Lecturer | In Service |
| 33 | NEBA nee NDOSIRI Bridget NDOYE | Senior Lecturer | In Service |

5- DEPARTMENT OF ORGANIC CHEMISTRY (CO) (34)

| | | | |
|----|---------------------------------|---------------------|-----------------------------|
| 1 | DONGO Etienne | Professor | VICE DEAN / DSSE |
| 2 | GHO GOMU TIH Robert Ralph | Professor | In Service |
| 3 | MBAFOR Joseph | Professor | In Service |
| 4 | NGADJUI TCHALEU B. | Professor | <i>Head of Dept. FMBS</i> |
| 5 | NGOUELA Silvére Augustin | Professor | In Service |
| 6 | NKENGFACK Augustin Ephraïm | Professor | Head of Department |
| 7 | NYASSE Barthélemy | Professor | <i>Director/UN</i> |
| 8 | PEGNYEMB Dieudonné Emmanuel | Professor | <i>Director/ MINESUP</i> |
| 9 | WANDJI Jean | Professor | In Service |
| 10 | Alex de Théodore ATCHADE | Associate Professor | <i>CD Rectorate/UWI</i> |
| 11 | FOLEFOC Gabriel NGOSONG | Associate Professor | <i>VICE DEAN Univ. Buea</i> |
| 12 | KEUMEDJIO Félix | Associate Professor | In Service |
| 13 | KOUAM Jacques | Associate Professor | In Service |
| 14 | MBAZOA née DJAMA Céline | Associate Professor | In Service |
| 15 | NOUNGOUE TCHAMO Diderot | Associate Professor | In Service |
| 16 | TCHOUANKEU Jean-Claude | Associate Professor | <i>VR/ UYII</i> |
| 17 | YANKEP Emmanuel | Associate Professor | In Service |
| 18 | TIH née NGO BILONG E. Anastasie | Associate Professor | In Service |
| 19 | MKOUNGA Pierre | Associate Professor | In Service |
| 20 | NGO MBING Joséphine | Associate Professor | In Service |
| 21 | TABOPDA KUATE Turibio | Associate Professor | In Service |
| 22 | KEUMOGNE Marguerite | Associate Professor | In Service |

| | | | |
|----|------------------------------|--------------------|------------|
| 23 | AMBASSA Pantaléon | Senior Lecturer | In Service |
| 24 | EYONG Kenneth OBEN | Senior Lecturer | In Service |
| 25 | FOTSO WABO Ghislain | Senior Lecturer | In Service |
| 26 | KAMTO Eutrophe Le Doux | Senior Lecturer | In Service |
| 27 | NGONO BIKOBO Dominique Serge | Senior Lecturer | In Service |
| 28 | NOTE LOUGBOT Olivier Placide | Senior Lecturer | In Service |
| 29 | OUAHOUE WACHE Blandine M. | Senior Lecturer | In Service |
| 30 | TAGATSING FOTSING Maurice | Senior Lecturer | In Service |
| 31 | ZONDENDEGOUNBA Ernestine | Senior Lecturer | In Service |
| 32 | NGOMO Orléans | Senior Lecturer | In Service |
| 33 | NGNINTEDO Dominique | Assistant Lecturer | In Service |

6- DEPARTMENT COMPUTER SCIENCE (IN) (25)

| | | | |
|----|------------------------------|---------------------|--|
| 1 | ATSA ETOUNDI Roger | Professor | Head of Department |
| 2 | FOUDA NDJODO Marcel Laurent | Professor | <i>Head of Dept ENS/Chef Div Sys.MINESUP</i> |
| 3 | TCHUENTE Maurice | Professor | <i>PCA UY II</i> |
| 4 | NDOUNAM René | Associate Professor | In Service |
| 5 | KOUOKAM KOUOKAM E. A. | Senior Lecturer | In Service |
| 6 | CHEDOM FOTSO Donatien | Senior Lecturer | In Service |
| 7 | MELATAGIA YONTA Paulin | Senior Lecturer | In Service |
| 8 | MOTO MPONG Serge Alain | Senior Lecturer | In Service |
| 9 | TINDO Gilbert | Senior Lecturer | In Service |
| 10 | TSOPZE Norbert | Senior Lecturer | In Service |
| 11 | WAKU KOUAMOU Jules | Senior Lecturer | In Service |
| 12 | TAPAMO Hyppolite | Senior Lecturer | In Service |
| 13 | ABESSOLO ALO'O Gislain | Assistant Lecturer | In Service |
| 14 | BAYEM Jacques Narcisse | Assistant Lecturer | In Service |
| 15 | DJOUWE MEFFEJA Merline Flore | Assistant Lecturer | In Service |
| 16 | DOMGA KOMGUEM Rodrigue | Assistant Lecturer | In Service |
| 17 | EBELE Serge | Assistant Lecturer | In Service |
| 18 | HAMZA Adamou | Assistant Lecturer | In Service |
| 19 | KAMDEM KENGNE Christiane | Assistant Lecturer | In Service |
| 20 | KAMGUEU Patrick Olivier | Assistant Lecturer | In Service |
| 21 | KENFACK DONGMO Clauvice V. | Assistant Lecturer | In Service |
| 22 | MEYEMDOU Nadège Sylvianne | Assistant Lecturer | In Service |
| 23 | MONTHÉ DJIADEU Valéry M. | Assistant Lecturer | In Service |
| 24 | JIOMEKONG AZANZI Fidel | Assistant Lecturer | In Service |

7- DEPARTMENT OF MATHEMATICS (MA) (35)

| | | | |
|----|--------------------------------|---------------------|---------------------------|
| 1 | BEKOLLE David | Professor | <i>Vice-Rector UN</i> |
| 2 | BITJONG NDOMBOL | Professor | <i>In Service</i> |
| 3 | DOSSA COSSY Marcel | Professor | In Service |
| 4 | AYISSI Raoult Domingo | Associate Professor | Head of Department |
| 5 | EMVUDU WONO Yves S. | Associate Professor | <i>CD/ MINESUP</i> |
| 6 | NKUIMI JUGNIA Célestin | Associate Professor | In Service |
| 7 | NOUNDJEU Pierre | Associate Professor | In Service |
| 8 | TCHAPNDA NJABO Sophonie B. | Associate Professor | Director/AIMS Rwanda |
| 9 | AGHOUEKENG JIOFACK Jean Gérard | Senior Lecturer | Chef Service MINPLAMAT |
| 10 | CHENDJOU Gilbert | Senior Lecturer | In Service |
| 11 | FOMEKONG Christophe | Senior Lecturer | In Service |
| 12 | KIANPI Maurice | Senior Lecturer | In Service |
| 13 | KIKI Maxime Armand | Senior Lecturer | In Service |

| | | | |
|---|---------------------------------|---------------------|---|
| 14 | MBAKOP Guy Merlin | Senior Lecturer | In Service |
| 15 | MBANG Joseph | Senior Lecturer | In Service |
| 16 | MBEHOU Mohamed | Senior Lecturer | In Service |
| 17 | MBELE BIDIMA Martin Ledoux | Senior Lecturer | In Service |
| 18 | MENGUE MENGUE David Joe | Senior Lecturer | In Service |
| 19 | NGUEFACK Bernard | Senior Lecturer | In Service |
| 20 | POLA DOUNDOU Emmanuel | Senior Lecturer | In Service |
| 21 | TAKAM SOH Patrice | Senior Lecturer | In Service |
| 22 | TCHANGANG Roger Duclos | Senior Lecturer | In Service |
| 23 | TCHOUNDJA Edgar Landry | Senior Lecturer | In Service |
| 24 | TETSADJIO TCHILEPECK M. E. | Senior Lecturer | In Service |
| 25 | TIAYA TSAGUE N. Anne-Marie | Senior Lecturer | In Service |
| 26 | DJIADEU NGAHA Michel | Assistant Lecturer | In Service |
| 27 | MBIAKOP Hilaire George | Assistant Lecturer | In Service |
| 28 | NIMPA PEFOUNKEU Romain | Assistant Lecturer | In Service |
| 29 | TANG AHANDA Barnabé | Assistant Lecturer | Director/MINTP |
| 8- DEPARTMENT OF MICROBIOLOGY (MIB) (13) | | | |
| 1 | ESSIA NGANG Jean Justin | Professor | DRV/IMPM |
| 2 | ETOA François Xavier | Professor | Head of Department Rector University of Douala |
| 3 | NWAGA Dieudonné M. | Associate Professor | In Service |
| 4 | NYEGUE Maximilienne Ascension | Associate Professor | In Service |
| 5 | SADO KAMDEM Sylvain Leroy | Associate Professor | In Service |
| 6 | BOYOMO ONANA | Associate Professor | In Service |
| 7 | RIWOM Sara Honorine | Associate Professor | In Service |
| 8 | BODA Maurice | Senior Lecturer | In Service |
| 9 | BOUGNOM Blaise Pascal | Senior Lecturer | In Service |
| 10 | ENO Anna Arey | Senior Lecturer | In Service |
| 11 | ESSONO OBOUGOU Germain G. | Senior Lecturer | In Service |
| 12 | NJIKI BIKOÏ Jacky | Senior Lecturer | In Service |
| 13 | TCHIKOUA Roger | Senior Lecturer | In Service |
| 9. DEPARTEMENT OF PHYSICS (PHY) | | | |
| 1 | ESSIMBI ZOBO Bernard | Professor | In Service |
| 2 | KOFANE Timoléon Crépin | Professor | In Service |
| 3 | NDJAKA Jean Marie Bienvenu | Professor | Head of Department |
| 4 | NJOMO Donatien | Professor | In Service |
| 5 | PEMHA Elkana | Professor | In Service |
| 6 | TABOD Charles TABOD | Professor | In Service |
| 7 | TCHAWOUA Clément | Professor | In Service |
| 8 | WOAFO Paul | Professor | In Service |
| 9 | EKOBENA FOU DA Henri Paul | Associate Professor | <i>Chef Division. UN</i> |
| 10 | NJANDJOCK NOUCK Philippe | Associate Professor | <i>Chef Serv. MINRESI</i> |
| 11 | BIYA MOTTO Frédéric | Associate Professor | DG/Mekin |
| 12 | BEN- BOLIE Germain Hubert | Associate Professor | CD/ENS/UN |
| 13 | DJUIDJE KENMOE épouse ALOYEM | Associate Professor | In Service |
| 14 | NANA NBENDJO Blaise | Associate Professor | In Service |
| 15 | NOUAYOU Robert | Associate Professor | In Service |
| 16 | SIEWE SIEWE Martin | Associate Professor | In Service |
| 17 | ZEKENG Serge Sylvain | Associate Professor | In Service |
| 18 | EYEBE FOU DA Jean sire | Associate Professor | In Service |

| | | | |
|----|-------------------------------|---------------------|-----------------------------|
| 19 | FEWO Serge Ibraïd | Associate Professor | In Service |
| 20 | HONA Jacques | Associate Professor | In Service |
| 21 | OUMAROU BOUBA | Associate Professor | |
| 22 | SAIDOU | Associate Professor | Sub Director/Minresi |
| 23 | SIMO Elie | Associate Professor | In Service |
| 24 | BODO Bernard | Senior Lecturer | In Service |
| 25 | EDONGUE HERVAIS | Senior Lecturer | In Service |
| 26 | FOUEDJIO David | Senior Lecturer | In Service |
| 27 | MBANE BIOUELE | Senior Lecturer | In Service |
| 28 | MBINACK Clément | Senior Lecturer | In Service |
| 29 | MBONO SAMBA Yves Christian U. | Senior Lecturer | In Service |
| 30 | NDOP Joseph | Senior Lecturer | In Service |
| 31 | OBOUNOU Marcel | Senior Lecturer | Chef Service /Univ Douala |
| 32 | TABI Conrad Bertrand | Senior Lecturer | In Service |
| 33 | TCHOFFO Fidèle | Senior Lecturer | In Service |
| 34 | VONDOU Derbetini Appolinaire | Senior Lecturer | In Service |
| 35 | WOULACHE Rosalie Laure | Senior Lecturer | In Service |
| 36 | ABDOURAHIMI | Senior Lecturer | In Service |
| 37 | ENYEGUE A NYAM épouse BELINGA | Senior Lecturer | In Service |
| 38 | WAKATA née BEYA Annie | Senior Lecturer | <i>Chef Serv. MINESUP</i> |
| 39 | MVOGO ALAIN | Senior Lecturer | <i>Sub Director/MINESUP</i> |
| 40 | CHAMANI Roméo | Assistant Lecturer | In Service |
| 41 | MLI JOELLE LARISSA | Assistant Lecturer | <i>In Service</i> |

10- DEPARTMENT OF EARTH SCIENCES (ST) (42)

| | | | |
|----|----------------------------|---------------------|--|
| 1 | NDJIGUI Paul Désiré | Professor | Head of Department |
| 2 | BITOM Dieudonné | Professor | <i>DEAN / FASA / UDs</i> |
| 3 | NZENTI Jean-Paul | Professor | In Service |
| 5 | KAMGANG Pierre | Professor | In Service |
| 6 | MEDJO EKO Robert | Professor | <i>Technical Adviser/UYII</i> |
| 4 | FOUATEU Rose épouse YONGUE | Associate Professor | In Service |
| 7 | NDAM NGOUPAYOU Jules-Remy | Associate Professor | In Service |
| 8 | NGOS III Simon | Associate Professor | CD/Uma |
| 9 | NJILAH Isaac KONFOR | Associate Professor | In Service |
| 10 | NKOUMBOU Charles | Associate Professor | In Service |
| 11 | TEMDJIM Robert | Associate Professor | In Service |
| 12 | YENE ATANGANA Joseph Q. | Associate Professor | <i>Chef Div. /MINTP</i> |
| 13 | ABOSSOLO née ANGUE Monique | Associate Professor | <i>Chef div. DAASR / FS</i> |
| 14 | GHOGOMU Richard TANWI | Associate Professor | CD/UMa |
| 15 | MOUNDI Amidou | Associate Professor | <i>Chef Div. MINIMDT</i> |
| 16 | ONANA Vincent | Associate Professor | In Service |
| 17 | TCHOUANKOUE Jean-Pierre | Associate Professor | In Service |
| 18 | ZO'O ZAME Philémon | Associate Professor | <i>DG/ART</i> |
| 19 | MOUNDI Amidou | Associate Professor | <i>Chef Div. MINIMDT</i> |
| 20 | BEKOA ETIENNE | Senior Lecturer | <i>In Service</i> |
| 21 | BISSO DIEUDONNE | Senior Lecturer | <i>Director/Projet Barrage Memve'ele</i> |
| 22 | ESSONO Jean | Senior Lecturer | <i>In Service</i> |
| 23 | EKOMANE EMILE | Senior Lecturer | <i>En pste</i> |
| 24 | FUH Calistus Gentry | Senior Lecturer | <i>Sec. D'Etat/MINMIDT</i> |
| 25 | GANNO Sylvestre | Senior Lecturer | In Service |
| 26 | LAMILEN BILLA Daniel | Senior Lecturer | In Service |
| 27 | MBIDA YEM | Senior Lecturer | <i>In Service</i> |

| | | | |
|----|------------------------------|--------------------|---------------------|
| 28 | MINYEM Dieudonné-Lucien | Senior Lecturer | <i>CD/Uma</i> |
| 29 | MOUAFO Lucas | Senior Lecturer | In Service |
| 30 | NJOM Bernard de Lattre | Senior Lecturer | In Service |
| 31 | NGO BELNOUN Rose Noël | Senior Lecturer | In Service |
| 32 | NGO BIDJECK Louise Marie | Senior Lecturer | In Service |
| 33 | NGUETCHOUA Gabriel | Senior Lecturer | CEA/MINRESI |
| 34 | NYECK Bruno | Senior Lecturer | In Service |
| 35 | TCHAKOUNTE J. épouse NOUMBEM | Senior Lecturer | <i>CT / MINRESI</i> |
| 36 | METANG Victor | Senior Lecturer | In Service |
| 37 | NOMO NEGUE Emmanuel | Senior Lecturer | In Service |
| 38 | TCHAPTCHET TCHATO De P. | Senior Lecturer | In Service |
| 39 | TEHNA Nathanaël | Senior Lecturer | In Service |
| 40 | TEMGA Jean Pierre | Senior Lecturer | In Service |
| 41 | MBESSE CECILE OLIVE | Senior Lecturer | In Service |
| 42 | ELISE SABABA | Senior Lecturer | In Service |
| 43 | EYONG JOHN TAKEM | Assistant Lecturer | In Service |
| 44 | ANABA ONANA Achille Basile | Assistant Lecturer | In Service |

Number of teachers of the Faculty of Science of the University of Yaoundé I

| NUMBER OF TEACHERS | | | | | |
|--------------------|---------------|-------------------------------|------------------|---------------------|-----------------|
| DEPARTMENT | Professors | Assistant Lecturer Professors | Senior Lecturers | Assistant Lecturers | Total |
| B.C. | 5 (1) | 10 (5) | 21 (10) | 3 (1) | 39 (17) |
| B.P.A. | 11 (1) | 9 (3) | 20 (8) | 3 (5) | 43 (17) |
| B.P.V. | 4 (0) | 9 (2) | 10 (2) | 4 (4) | 27 (8) |
| C.I. | 10(1) | 8(2) | 16 (4) | 0 (2) | 34 (9) |
| C.O. | 9 (0) | 13 (3) | 8 (2) | 1 (0) | 31 (5) |
| I.N. | 3 (0) | 1 (0) | 8 (0) | 12 (3) | 24 (3) |
| M.A. | 3 (0) | 5 (0) | 18 (1) | 4 (0) | 30 (1) |
| M.B. | 2 (0) | 5 (2) | 6 (2) | 0 (0) | 13 (4) |
| P.H. | 8 (0) | 17 (0) | 15 (2) | 2 (1) | 42 (3) |
| S.T. | 5 (0) | 15 (2) | 23 (3) | 2 (0) | 45 (5) |
| Total | 60 (3) | 92 (19) | 145 (34) | 31 (16) | 328 (72) |

Total 328 (72)

Distributed as follows :

- Professors **60 (3)**
- Associate Professors **92 (19)**
- Senior Lecturers **145 (34)**
- Assistant Lecturers **31 (16)**

() = Number of Women

Contents

| | |
|---|-----------|
| Abstract | i |
| Acknowledgements | ix |
| I PREAMBLE | 1 |
| 1 Introduction | 2 |
| 1.1 Motivation and Objectives | 2 |
| 1.2 Illustrative Example | 4 |
| 1.3 Contributions | 6 |
| 1.4 Publications | 8 |
| 2 Background | 9 |
| 2.1 Business Process Management | 9 |
| 2.1.1 The Process Management Spectrum | 9 |
| 2.1.2 BPM Techniques | 11 |
| 2.2 Case Management and Dynamic Processes | 12 |

| | | |
|-------|--|-----------|
| 2.2.1 | Case Management | 12 |
| 2.2.2 | Artifact-Centered Business Processes | 14 |
| 2.3 | Collaboration and User Interactions | 15 |
| 2.4 | Flexibility in BPM | 16 |
| 2.4.1 | Types of process flexibility | 17 |
| 2.5 | Disease Surveillance | 18 |
| 2.5.1 | Definition | 18 |
| 2.5.2 | Human Centeredness | 19 |
| 2.5.3 | Data Drivenness | 20 |
| 2.5.4 | Timeliness and Uncertainty | 21 |
| | Conclusion | 22 |
| | II GUARDED ATTRIBUTE GRAMMARS | 23 |
| | Introduction | 24 |
| | 3 Guarded Attribute Grammars - GAG | 26 |
| 3.1 | A Grammatical Approach to Task Decomposition | 26 |
| 3.1.1 | Modelling Attributes | 28 |
| 3.1.2 | Modelling Tasks | 31 |
| 3.2 | Syntax of a Guarded Attribute Grammar | 33 |
| 3.3 | Behaviour of a Guarded Attribute Grammar | 37 |

| | | |
|------------|---|-----------|
| 3.3.1 | Configuration of a Guarded Attribute Grammar | 38 |
| 3.3.2 | Atomic Step - Applying a Business Rule | 40 |
| 3.4 | Examples | 45 |
| 4 | Composition, Distribution, and Soundness of GAGs | 51 |
| 4.1 | Composition of Guarded Attribute Grammars | 51 |
| 4.2 | Distribution of a Guarded Attribute Grammar | 56 |
| 4.3 | Soundness of Guarded Attribute Grammars | 64 |
| 4.3.1 | Preliminaries | 66 |
| 4.3.2 | Building a Complete-Artifact | 69 |
| 4.3.3 | Checking Soundness of a Complete Artifact | 71 |
| | Conclusion | 73 |
| III | THE ACTIVE WORKSPACE FRAMEWORK | 75 |
| | Introduction | 76 |
| 5 | Active Workspaces for User-Centered, Distributed Collaborative Systems | 78 |
| 5.1 | The Active Workspaces Model | 78 |
| 5.1.1 | Services and Roles | 79 |
| 5.1.2 | Flexible Process Enactment and Incremental Design | 81 |
| 5.1.3 | Relationships between Tasks | 83 |
| 5.2 | Timed Guarded Attribute Grammars | 85 |

| | | |
|----------|---|------------|
| 5.3 | Collaboration and User Interactions | 86 |
| 5.3.1 | Requesting or Assigning Work | 87 |
| 5.3.2 | Artifact Sync and Migration | 90 |
| 5.4 | Towards a Language for Active-Workspace Specification | 92 |
| 5.4.1 | Enriching Guards and Semantic Rules | 92 |
| 5.4.2 | An Extended Syntax for GAGs | 95 |
| 5.4.3 | An Example - Flu Outbreak Surveillance | 102 |
| 6 | An Active-Workspaces Prototype and Example | 107 |
| 6.1 | Description and Architecture | 107 |
| 6.1.1 | The Active-Workspace Server | 109 |
| 6.2 | A Domain Specific Language for GAG Specification | 110 |
| 6.2.1 | Domain Specific Languages | 110 |
| 6.2.2 | GAG-DSL Generalities | 112 |
| 6.2.3 | The GAG-DSL Syntax | 117 |
| 6.2.4 | The GAG-DSL Operational Semantics | 127 |
| 6.3 | Enactment User Interface and Actions | 128 |
| 6.4 | GAG-DSL Example: Flu Outbreak Management | 129 |
| | Conclusion | 138 |

| | |
|---|------------|
| IV Conclusion | 139 |
| 7 Conclusion, Discussion, and Future Works | 140 |
| 7.1 Assessment of the Model | 141 |
| 7.2 Future Work | 143 |
| Bibliography | 145 |
| Annex | 158 |

Part I

PREAMBLE

Chapter 1

Introduction

The need to work together doing more with less, to meet unyielding deadlines, to deal with unprecedented levels of uncertainty, and to do it all flawlessly has become the norm, not the exception [14].

1.1 Motivation and Objectives

A few decades ago, the need to digitally support enterprise activities, improve on their organization, visibility, control, and maintenance was most felt and several research communities emerged each handling different aspects of activity modelling and automation. The objective being to clearly identify the different activities as a sequence of inter-connected actions (a process) and capture these into off-the-shelf solutions that are usable in enterprise environments.

The Business Process Management (BPM) [34] domain was created to address these concerns in the face of fast evolving technologies and with the ever-increasing complexity in enterprise activities and customer demands. BPM solutions aided companies not only to cope with this increasing complexity, but also to improve their productivity, reduce costs, and to stay competitive [124].

A Business Process is defined as a set of activities whose execution realises a certain goal [124]. It is the key to organising activities, improving and understanding their inter-dependencies. Business Process Management on the other hand brings together the concepts, methods and techniques necessary to support the design, administration, configuration, and analysis of business processes[124, 34].

From its definition, it is clear that BPM is well adapted to model structured processes: business processes that can be completely defined prior to their execution [119]. Precisely, BPM divides

system modelling into four major phases: *i. the design analysis phase* during which a succinct description of activities as well as the required data and resources are identified from requirements and expressed in a suitable modelling language; *ii. the configuration phase* during which a process aware information system (PAII) is configured to support execution of the designed process on some execution architecture; *iii. the enactment and monitoring phase* during which process instances are created and run using the configured PAII; *iv. the diagnosis phase* during which enacted processes are analysed to identify problems and aspects that should be improved upon [119, 114].

BPM therefore supports "process-aware" systems, i.e., systems in which information about the operational processes can be made explicit during the design phase.

We note however that this approach becomes insufficient to capture today's working environments: the increased need to innovate and respond to unpredictable (emerging) process needs in near real time, the need for knowledge based (including data and users) process orchestration, the support for collaborative work and decision making in uncertain contexts.

While most formerly structured systems are tilting towards more flexible process management solutions to improve their productivity in the face of competition, other systems are by definition mostly unpredictable and cannot be completely modelled using BPM techniques. This is the case with the **Disease Surveillance Process**.

In disease surveillance, several geographically distant users with heterogeneous profiles collaborate and share information extensively over a highly cognitive and unpredictable process [90] in order to detect and/or pre-empt the occurrence of disease outbreaks and take timely actions [85]. Based on the classification in [114], disease surveillance can be considered a semi-structured process since its high level activities and their sequencing are known at design time but the actual low level activities as well as their execution order only become known at runtime as communication between users intensify and new information becomes available.

These new requirements bring to light new modelling objectives: supporting all forms of work (structured, unstructured, ad-hoc, and context sensitive work [14]), using data to dictate the choice and ordering of activities, giving users higher expressive power in process design and execution, and supporting all forms of interactions between users.

We focus our research on these new requirements, the modelling of the so-called dynamic (adaptive) processes, which are data-centric and user-driven: data plays a key role in the choice and ordering of work and users interact extensively over process models defined almost on-the-fly and executed in (time) constrained environments. Examples of activity domains requiring a dynamic process model include:

- **Medical Consultations:** the set of activities the doctor performs, and the order of execution depend highly on the signs and symptoms, the patient's history, on laboratory

analysis results, on the doctors expertise and experience from similar cases in the past, etc. The activities are thus likely to be different on each case, and still, when they remain same, the physician may not always do them in the same order all the time.

- **Crisis Management:** there exist guidelines for almost every form of crisis. However, crises are mostly spontaneous and there are hardly ever all the necessary resources required to execute the prescribed guidelines. Also, guidelines are indications of what should be done, the actual activities, usually performed by unskilled volunteers, are determined on site and effective coordination is required for efficiency.
- **Disease Outbreak Investigation:** outbreak investigation has been characterized as being a collaborative expert activity involving several cognitive tasks with several decision points, on data built progressively [91]. It therefore requires modelling that guarantees considerable flexibility and aids the user to implement appropriate contextualized activities rather than predefined (and sometimes inappropriate) ones.

The actual models for these processes are built on a per-case basis and we talk of Case Management [104]. Case Management is an emerging way to support users (knowledge workers) in applications that require a level of flexibility beyond the process flows of classical BPM [70]. It is strongly centered around data: it organises processes around case folders – that hold business documents and other information – as the primary building block for managing processes [70]. The Case Management Model and Notation (CMMN), the Object Management Group (OMG) standard for case management, promotes the notion of business artifacts [78, 62], which are holistic representations of data and processes. A business artifact has an information model and a lifecycle model. The so-called business rules expressed on the information model are used to control the evolution of the artifact’s lifecycle [78].

1.2 Illustrative Example

Foodborne disease outbreak Investigation:

The following scenario describes a response to a foodborne disease outbreak. It is based on a real-world case that occurred in a French military base. The outbreak is declared by the physician at the emergency unit of the military base and responded to by the surveillance unit of the said base. The physician receives within a day, nine (9) patients with abdominal pains, fever, and nausea. He notices that all 9 persons belong to the same military unit had all been at a social gathering the previous night with their families. He immediately reports the suspected outbreak to the surveillance unit while waiting for laboratory analysis results to ascertain which bacteria caused the poisoning. The epidemiologist at the surveillance unit contacts the affected military unit to obtain the list of all attendees of the gathering, the kitchen personnel, and the list of all food items

consumed. He initiates a survey on the attendees and within 3 days the survey results showed that amongst the 40 persons (military personnel) who attended the gathering, 25 fell sick. Also, these 25 persons ate 4 food items in common. The physician eventually receives the laboratory tests and sends them to the epidemiologist. The results show that the bacteria that caused the poisoning was *Shigella Sonnei*. With this new information, the epidemiologist sends the food samples to the laboratory in a bit to know which of the items contained the bacteria. He also starts preparing to carry out an investigation to trace the origin of the bacteria. The laboratory results showed that the beef consumed at the gathering contained the bacteria. With this information the epidemiologist engages the veterinarian with whom he identifies the slaughter house and eventually run more tests on animal food and water detect the primary source. In the mean time, all patients are placed under treatment and the civilian surveillance unit notified and the list of civilian attendees of the gathering sent to them.

In this scenario, we illustrate how data is used to dictate what can be done at any given moment. The general guidelines for a foodborne disease outbreak investigation prescribe a survey to identify the food item and another survey to identify the source of the bacteria. Both surveys proceed in parallel but as we see in the scenario, the actions that need to be carried out in each of these surveys as well as the actors that will intervene only become known and defined when data becomes available. For instance, the information that the bacteria at the origin of the outbreak was in the meat will greatly influence the actions that need to be taken and the choice of the users to include in the survey.

Flu Outbreak Investigation:

Let us now describe another scenario of Flu Outbreak surveillance and investigation which illustrates the need to support automated work, adhoc work, and manual work and which lays emphasis on the implication of users to drive process enactment. In Flu surveillance, every physician who notices Flu symptoms in a patient registers the patient in the Flu suspect list which he declares to the surveillance center in the weekly Flu report. This information is entered into a database on which automated outbreak surveillance algorithms run. These algorithms search for an abnormal increase in Flu cases in a population within a period, and at a given place – a Flu epidemic. When such an abnormal increase occurs, an alarm is produced, and the head epidemiologist notified. The objective of the epidemiologist is to confirm the outbreak alarm and hence produce an outbreak alert or to discard the alarm [19]. If for example a similar alarm had been produced the previous week, he can safely discard the new alarm since it most likely concerns the current epidemic. If on the other hand the alarm is new, the epidemiologist initiates situational diagnosis [19, 108] during which he will combine biological data, environmental data, and other sources of information to decide whether the outbreak alarm is real. To do so, he creates several (adhoc) tasks to request precise additional information, give directives, or simply inform. For instance, he can create a task for

each of the reporting physicians in the alarm data requesting that they provide him with additional diagnostic information: lab results, medical history, places the patients had been the weeks before the onset of symptoms etc. This information might have been registered during the patient visits but not reported. When the epidemiologist receives this information, he can define a specific *case-definition* and send as directive to all hospitals, clinics, laboratories, pharmacies, etc. in the affected area, requiring them to report any cases that fall in the case-definition criteria and to proceed systematically with sample extraction and laboratory analysis for all such cases. Rich with this information, the epidemiologist might be able to decide on the validity of the alarm or might decide to create more tasks requesting more information or using automated data analysis tools to obtain more insight into the alarm.

This scenario shows how the user actively drives process enactment based on the available data. It illustrates the semi-structuredness of the disease surveillance process, with automated predefined segments and ad-hoc segments that only become discovered as data is produced and whose effective inclusion in the process rests on decisions made by the user at run-time.

The disease surveillance process can be characterised as being **data-centered** and **user-driven** involving several **collaborating users**. It requires modelling which expresses the essential flexibility required by human case workers for: (i) run-time definition, selection, and planning of tasks, (ii) run-time ordering, (iii) ad-hoc collaboration with other case workers, and (iv) (Multi-user) decision making support.

Our objective in this thesis is to propose an Adaptive Case Management model for distributed collaborative systems. In the next section, we briefly discuss our contributions and elucidate its principal properties.

1.3 Contributions

1. AW based on GAGs

We design a distributed data-driven and user-centered case management system, evocative of artifact-centric approaches but with emphasis on organising the work on each user's workspace and providing tools to enhance flexible and improved expressiveness in task resolution. Our model is built on two foundational concepts: Guarded Attribute Grammars and Active-Workspaces.

(a) **Guarded Attribute Grammars** *Chapters 3-4:*

We assume the widely studied hierarchical task decomposition technique for task analysis [102], then we express the task hierarchies as rewriting rules and propose a

formal specification based on productions of a (guarded) attribute grammars. The attributes attached to the (sorted) grammar symbols in each production are used by implicit semantic rules to pass down data from a task to its subtasks and to compute the output of a task from the outputs of its subtasks.

The GAG model is declarative, modular, distributed, user-centered, and data (artifact) driven formal specification language for dynamic processes.

One key property of process modelling techniques is *soundness*, that is, the ability of every case to be executed to completion no matter how it started or what data it was initialised with. We define a restriction on guarded attribute grammars and describe a procedure that verifies that such GAGs terminate and are sound.

(b) **Active-Workspaces** *Chapter 5:*

Our case management model is centered on the notion of users and users' workspace. A user's workspace comprises a GAG specification of the services he offers and several mindmaps (trees) used to visualize and organize the work carried out by the user as well as the information used by and/or produced as a result of the resolution of the task. Each of the mindmaps correspond to a service offered by the user and each in turn might contain several artifacts – concrete instances of the corresponding service –, initiated either by the user or as a result of a request from a distant user, in which case, the results output will have to be sent to the distant user.

The internal nodes (closed nodes) of an artifact represent tasks for which a resolution method has been assigned, while the leaf nodes (open nodes) represent pending tasks. Data arriving at open nodes is used by the operational semantics of GAGs to automatically filter doable actions at these nodes. If more than one action is possible, then the user has to choose which to perform. If none of the doable actions suit the pursued objective, new ones can be added by defining new grammar productions in the underlying GAG. Actions correspond to guarded attribute grammar productions which when applied develop the artifact tree further and creates new open nodes. The active workspace model gives users run-time control over process design and enactment, hence the name **Active-Workspace**.

2. **AW-Framework: Prototype & Example** *Chapter 6*

To demonstrate the properties of the AW/GAG model, we design a prototype for the model in what we call the AW-Framework. This framework includes, a language syntax for GAG specification (user, data, task, and interaction modelling), an implementation of the language as an internal domain specific language into Haskell (a general purpose purely functional language), and a graphical user interface for process enactment and simulation of distributed execution. We also model a complete example of a disease surveillance system using the AW-Framework.

1.4 Publications

Journal Paper

1. Eric Badouel, Loïc Hélouët, Georges-Edouard Kouamou, Christophe Morvan, and Robert Fondze Jr Nsaibirni. Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars. *ACM SIGAPP Applied Computing Review*, 15(3):6–34, 2015

Conferences

1. Robert Fondze Jr Nsaibirni, Eric Badouel, Gaëtan Texier, and Georges-Edouard Kouamou. Active-Workspaces: A Dynamic Collaborative Business Process Model for Disease Surveillance Systems. In *HIMS'16 - The 2nd International Conference on Health Informatics and Medical Systems, Las Vegas, USA*, 2016
2. Robert Fondze Jr Nsaibirni and Gaëtan Texier. User Interactions in Dynamic Processes: Modeling User Interactions in Dynamic Collaborative Processes using Active Workspaces. In *Proceedings of CARI'16*, pages 109–116, Hammamet, Tunisia, 2016
3. Robert Fondze Jr Nsaibirni, Gaëtan Texier, and Georges-Edouard Kouamou. Modelling disease surveillance using Active Workspaces. In *Conference de Recherche en Informatique, Yaoundé, Cameroon*, 2015

Chapter 2

Background

In the introductory chapter, we situated the context and requirements for modelling dynamic collaborative processes. In this chapter, we expatiate upon the underlying concepts and present a state of the art related to each of them. Then we present an analysis and characterization of the disease surveillance process as a dynamic process.

2.1 Business Process Management

2.1.1 The Process Management Spectrum

In [31], Di Ciccio et al. presents a classification of business processes based on the required degree of flexibility and predictability. Their classification is similar to previous classifications in [114, 57]. As the required degree of flexibility increases, it becomes more and more difficult to automate and control business processes.

In *Structured Processes* the process logic is known in advance and the activities, dependencies, and associated resources are pre-definable end-to-end. They can be repeatedly instantiated in a predictable and controlled manner. Examples are factory manufacturing and administrative procedures.

Structured processes with ad-hoc exceptions are structured processes augmented with exception handlers for anticipated exceptions. Such exception handlers are: undo/redo, skip, etc. Unanticipated exceptions often require redesigning the business process.

In *Unstructured processes with pre-defined segments*, the overall process logic is unknown, but policies and regulations can be used a-priori to define structured fragments. These fragments are then incorporated into the overall process at runtime on a per-case basis.

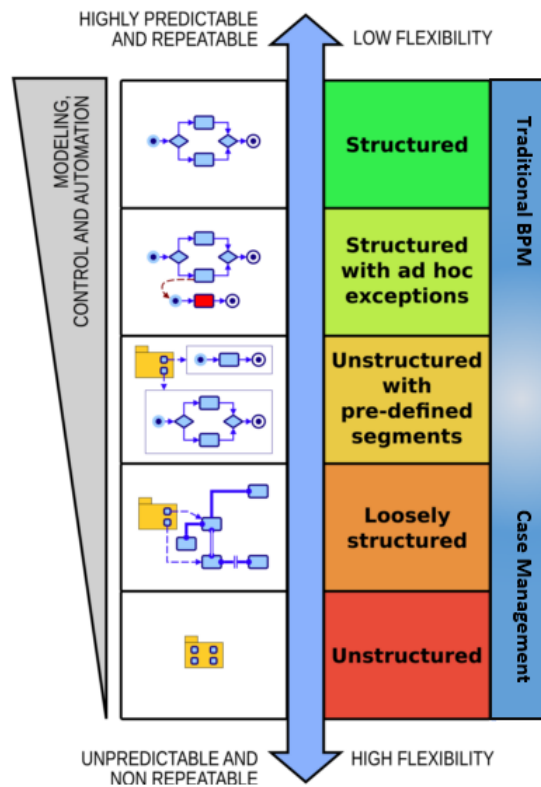


Figure 2.1 – Process Management Spectrum [31]

Loosely structured processes have a known set of possible activities, but it is unknown when each will be executed due to the indefinite number of foreseeable futures. For instance, the actions to take to investigate an outbreak alarm may be known but when and how they will be executed depends on the alarm data.

Lastly, *unstructured processes* are those for which no predefined model can be defined and little or no automation is possible. Users rely on experience to perform ad-hoc tasks on a per-case basis. Only the pursued goal is known a priori.

As we move down the spectrum, we lose in automation and control and gain in flexibility and unpredictability. Traditional BPM techniques are adapted for structured and fairly structured application domains. Unstructured processes on the other hand require more data centric and rule-based approaches [31], which are the basis of case management [104]. The latter are highly knowledge intensive processes: processes requiring a modelling technique that does not follow the design – execute & monitor – analyse – re-design paradigm [114] but supports a continuous interleaving and overlapping between design, execution and adaptation phases [30, 31].

2.1.2 BPM Techniques

A chronological evolution is apparent from the development of process modelling languages and methods. At the onset is activity centric methods which lay emphasis on the flow of activities (constrained or not), then comes data centric methods which use data to dictate when and how a process evolves, and finally & most recently, artifact centric methods which describe entities called artifacts to handle data and flow holistically so that both evolve as a function of the other. In the following section, we present a summary of the techniques developed in the literature for activity centric business processes (workflows).

2.1.2.1 Activity Centric Workflows

Organized around activity flows, with tasks and operations as the core building blocks, the objective of activity centric workflows is to execute a set of operations in a predefined order. The manipulated data is not explicitly considered, it is regarded as a second-class citizen.

Petri nets [77, 112] are the oldest and most investigated process modelling language with a simple and intuitive graphical notation. Their precise semantics make them well suited for formal analysis in general and process model analysis in particular. In process models, places represent states and/or conditions while transitions represent events or tasks, and tokens represent information objects. Richer process specifications are obtained with coloured Petri-nets when semantic meaning and time values are attached to the colour of the token and the tokens respectively.

Workflow nets - WF-nets [112] are enhanced Petri nets for workflow modelling. They include new concepts and notations which allow for easier specification of complex business processes. In particular, WF-nets have a start and an end place, and every other place is situated along the path from start to end. They can express exclusive and parallel behaviour using split and join nodes and the tokens carry data specific to process instances. Hence several instances of a process can coexist.

Yet Another Workflow Language - YAWL [117]. Developed as a formal syntax (set theory and predicate logic) and semantics (coloured Petri nets) for the large collection of workflow patterns [118] that were identified by the workflow patterns initiative¹.

Business Process Modelling Notation - BPMN [83]. A graph based modelling notation developed under the coordination of the Object Management Group (OMG) with over 50 distinct graphical modelling constructs. It combines in a single language constructs and best practices of the other existing approaches from the business to the technical levels of abstraction. It

1. Workflow Patterns Initiative: <http://www.workflowpatterns.com>

contains atomic activities (tasks) which can be logically nested using gateways, message and control flows and can be used to express very complex process models.

The BPMN execution semantics is informally based on tokens, that is, data is used at gateways to direct the execution of the process instances or as prerequisites for tasks. Data modelling is however optional making BPMN a model where data objects play a supporting role to the control flow.

Event-driven Process Chains - EPCs [121]. They provide a classical informal graph-based notation for process specification widely used in industry. Nodes in EPC graphs represent either activities, events, or connectors while edges represent the control flow. EPC schema are not directly executable and since they provide a limited set of control flow constructs, industry demanded for extended EPCs (eEPCs) that are augmented with capabilities to model data objects, organizational entities and interfaces to processes. Data is however not directly modelled on the flow graph (extra nodes are added to represent data objects) hence no support for the data life cycle is provided.

2.2 Case Management and Dynamic Processes

One key characteristic of dynamic processes is their collaborative (cooperative) nature. They generally involve several participants with different skills from different domains at different levels, hence the resulting processes may include innovative and creative parts which are not easily straitjacketed into classical control-based process models [31]. A dynamic process is therefore a knowledge intensive process which according to [46] is one whose value can only be created through the fulfilment of the knowledge requirements of the process participants.

In [111], knowledge intensive processes are defined as those whose conduct and execution are heavily dependent on knowledge workers performing various interconnected knowledge intensive decision-making tasks. They are genuinely knowledge, information and data centric and require substantial flexibility at design- and run-time. This definition stresses the need to place knowledge and knowledge workers at the center of process modelling. Knowledge is used in dynamic processes not only as operational data (to compute required output) but also to control and/or constrain the execution (the when, the how, and the by whom) of the entire process.

2.2.1 Case Management

Case management [70, 104] or Case Handling [120] was conceived to handle work in application domains that require considerable levels of flexibility and knowledge-worker's expressive power.

Van der Aalst and Weske [120] described it as *a new paradigm for supporting flexible and knowledge intensive processes*. Also, market analysts at Forester describe case management as *highly structured, collaborative, dynamic, and information-intensive, driven by outside events and requiring incremental and progressive responses from the business domain handling the case* [64].

[104, 114, 31] suggest that having data centered process models greatly enhances flexibility while expressiveness is best achieved by the use of declarative rather than imperative modelling languages. A study presented in [31] identifies the major case management use cases as centered around being able to (i) defer the modelling of data, business rules (constraints on data), tasks (knowledge actions enriched with business rules), goals, knowledge workers, and events from design to run-time, and (ii) provide tools for explicit modelling of dependencies, interactions, and flexible chaining of these constructs to realise particular business goals.

A *case* is how a knowledge worker visualizes and organizes work (the coordination of multiple tasks, planned and unplanned, for a specific purpose). It contains data about the case and is represented by a business artifact which carries all of the information that is required and/or generated through the processing of the case [14]. Examples of cases include: a patient record, an outbreak alarm, an insurance claim, a natural disaster investigation, a laboratory test request, a lawsuit, etc. Also, a case can contain sub-cases. For example, in an outbreak alarm investigation case, several sub-cases might be created based on the type of outbreak, and these sub-cases in turn might initiate several laboratory analysis requests cases.

Adaptive (or dynamic) case management (ACM) [14, 104] seeks to support the management of work the way knowledge workers know it must be done [14]. More precisely, dynamic case management ensures that all forms of work (*automated work, manual work done by users, ad hoc work, and knowledge sensitive work*) are supported by case management systems [14].

The intended flexibility in case management is achieved through the use of the so-called case-folders which hold all the required business documents and other information needed to handle the case, and on which a behaviour model is assigned to guide its evolution in the business process [70]. The notion of Business Artifacts [78, 62] was introduced a couple of decades ago and promulgated recently by the CMMN (Case Management Model and Notation) [71] to enhance the need to place both data and control as first class citizens in process modelling techniques. Business artifacts are the basis on which researchers are building case management systems with the objective of obtaining formal verifiable models while preserving the flexibility and expressiveness properties of dynamic processes.

A business artifact has an information model and a lifecycle model [78]. The information model holds all operational data needed to process the case, while the lifecycle model describes the different ways the case can evolve through the business process.

2.2.2 Artifact-Centered Business Processes

Introduced in [120, 12] this approach is well suited for adaptive case management [120] and is centered around the notion of business artifacts that are holistic representations of data and processes. In [78], Nigam and Caswell define an artifact as a concrete, identifiable, self-describing chunk of information that can be used by a business person to actually run a business.

In [51] and [25] the artifact-centric approach is described as a workflow model comprising the following three dimensions: (i) *business artifacts* which are business relevant entities, (ii) *life cycles* which capture the end to end processing of business artifacts, and (iii) *business rules* expressed as constraints on the information model of business artifacts, used to control the evolution of the business artifacts in their lifecycles. Initial research on business artifacts modeled the lifecycles using variants of finite state machines [62, 61, 78, 54, 41, 15], Petri nets [67], or logical formulas depicting legal successors of a state [25]. However, these state-based formalisms do not provide the required flexibility since the specifications remain imperative and the designer has to explicitly describe how the process will unfold. Recent research has focused on more declarative lifecycle models [52, 25] which used rule-based formalisms to describe what activities are allowed or disallowed at any instant in the artifact lifecycle.

2.2.2.1 Finite State Machine based Lifecycles

The idea to associate a state machine with a business artifact was first introduced in the original ADocs model [62] and in [61], Kumaran et. al. developed a much more explicit association with activities represented as annotations on transitions between states. In the Business Entity Lifecycle Analytics (BELA) method [103], the states are thought of as milestones, that is, business-relevant operational objectives that a business artifact may achieve.

Another related model to business artifacts is the PHILharmonicFlows [63] model which enable a strong integration between process and data, supporting business objects with finite-state machine-based lifecycles.

2.2.2.2 Declarative Lifecycles

Contrary to the commonly used imperative paradigm of process modelling, the declarative approach is suitable for flexible processes that do not require a strict order of activities [38, 33, 32]. It limits their behavior by using constraints that sanction the starting and termination of tasks.

In 2010, IBM researchers introduced the concept of Business Entities (with Guard Stage Milestone (GSM)) Lifecycles (BEL) [52, 53, 25]. The idea is to replace the finite state machine-based life cycles with more declarative and intuitive lifecycles with the rule-based operational semantics expressed as event-condition-action (ECA-like [87]) rules. The GSM model has been adopted as the basis of the OMG standard Case Management and Notation (CMMN). Just like artifacts, BEL entities have an information model and a lifecycle model. The GSM life cycle meta-model allows for dynamic creation of subtasks and handles data attributes. Milestones correspond to business goals, stages contain one or more tasks whose execution aims at achieving one or several milestones, and guards are conditions or events which when achieved or triggered, activate stages and/or realize milestones.

However, interactions with users are modelled as incoming messages from the environment or as events from low-level (atomic) stages. In this way, users do not contribute to the choice of a workflow for a process. Also, the semantics of GSM model is given in terms of global snapshots, events can be handled by all stages as soon as they are produced, and a guard of a stage can refer to attributes of distant stages. Thus, this model is not directly executable on a distributed architecture. As reported in [37], distributed implementation may require restructuring the original GSM schema and relies on locking protocols to ensure that the outcome of the global execution is preserved.

2.3 Collaboration and User Interactions

Collaboration is working together with a single shared goal. It is a more aligned form of *cooperation* which describes performing together while working on "selfish" yet common goals. For instance, in a disease surveillance system, depending on the angle of view, the actors can be seen either as collaborating or cooperating. If we ignore every other complementary system (patient follow-up, research, etc) and consider only the objective of monitoring diseases for outbreaks, then different stakeholders will be seen as collaborating with the goal of detecting disease outbreaks. If on the other hand we view these different actors as going about their separate objectives while participating in disease surveillance, then they cooperate. In the remainder of this document, we interchangeably employ the two terms to mean the same thing.

Collaboration helps individual users and teams to harness their true potentials and break new grounds in efficiency and productivity, and technology plays an enabling role. *Collaborative software* or *groupware* is a set of computer tools that enhance collaboration between users. Groupware can be placed into three categories: (i) communication: that enhance structured and/or unstructured communication between users, (ii) deliberation or conferencing: that enhance brainstorming and collaborative decision making, (iii) coordination: running complex

interdependent (human-centered) activities towards a shared goal. In BPM and hence case management systems, the objective is to design tools for the third category.

Collaboration is an indispensable part of human-centered processes because it enhances business processes that not only depend on users but depend on users working together on specific activities to achieve set goals. Such users are geographically distant and interact in myriad ways. Several works have addressed collaboration concerns in BPM and case management systems [35, 69, 11, 10, 5]. In [5], it is argued that having a well-defined organizational context integrated into an electronic system (for transparency to users/actors) greatly increases the visibility users have of the processes they partake in.

In our work, we are interested in a collaborative system that helps users interact with each other and with the system to have work done. In the former case, they request for and provide help to each other while in the latter case, they flexibly build and enact processes together to resolve real world problems. Our declarative approach to process modelling favours modularity in process models, which is useful in distributed collaborative settings.

2.4 Flexibility in BPM

Flexibility is defined in [98] as the ability to deal with foreseen and unforeseen changes by varying or adapting those parts of the business process affected by them. Being able to adapt process models and enacted instances at design and/or runtime are fast becoming the rule rather than the exception in Business Process Modelling in general and especially in case management and dynamic process modelling. This is attributed to the continuous advances in domain knowledge, the increase in expert knowledge, and the diverse and heterogeneous nature of contextual variables. In such processes, several users with possibly heterogeneous profiles collaborate to achieve set goals on a process mostly designed on-the-fly.

Flexibility concerns has been widely studied in the BPM community [98, 3, 35, 92, 93, 114, 22] mostly focusing on process aware information systems (PAISs) and on declarative specifications. In [98], substantiated by [114], a comprehensive taxonomy for the different types of process flexibility is proposed, in a bit to curb the standardization gap and non-uniformity that exist amongst researchers. The proposed types (flexibility by definition, deviation, under-specification, and change) are designed to be independent of each other and all concur to improve the ability of business processes to respond to changes in their operating environment without necessitating a complete redesign of the underlying process model. However, they differ in the timing and manner in which they are applied. In the following section, we present this taxonomy from a dynamic process perspective.

2.4.1 Types of process flexibility

1. Flexibility by definition/design

In flexibility by definition, all possible changes are enumerated and incorporated into the process model at design time. It is achieved by incorporating support for workflow patterns (parallelism, choice, iteration, interleaving, multiple instances, cancellation, etc.) in the model. In classical process-centric models, achieving this form of flexibility is either not feasible because it is impossible to enumerate all situations or leads to highly complex models. In declarative constraint-based models for dynamic processes on the other hand, this is essentially achieved by using weaker constraints or reducing existing constraints.

2. Flexibility by deviation

In flexibility by deviation, the process model allows for runtime deviation from prescribed process model, which serves as an execution guide not as an imposed sequence. For instance, in a process model in which the physician registers a patient then examines him/her, the physician is likely to encounter situations where he is obliged to swap the *register* and *examine* tasks.

Again, this form of flexibility is incorporated into the model. This is achieved by providing support for undo/redo operations, skipping or inhibiting, bypassing or blocking, allowing the creation of additional instances of running instances, and allowing for arbitrary task invocation.

Dynamic knowledge intensive process generally don't have any predefined (prescribed) ordering and their declarative data centric models ensure that the ordering is discovered based on the data values and how they are evaluated in the constraints. This notwithstanding, providing support for undo/redo, inhibiting, and blocking operations improve the flexibility of the process model.

3. Flexibility by underspecification

This form of flexibility is adapted for semi-structured and/or adhoc-structured systems where what needs to be done at some **specific points** in a process only becomes apparent during execution or in collaborative systems where different parts of the process model need to be completed by separate users or groups of users. Such specific points are known in advance and marked as unspecified (using place-holders) to be filled subsequently at runtime.

Again, this form of flexibility requires that such points be identified at design time and filling up place-holders occurs in any of two ways: either by using a predefined process fragment (late binding) or defining one from scratch (late modelling).

Allowing for the execution of (well-formed) incomplete process specifications is an asset for every dynamic process modelling tool. It delays modelling till what should be done is known.

4. Flexibility by change

In human centered dynamic process models, it is assumed that two users in the same role do not necessarily proceed in the same way to resolve a given task. In the worst-case scenario, each of the users employ a different process model on each case - the user redefines the process entirely based on the case before him.

Flexibility by change is the ability of process models to accommodate on-the-fly change to the underlying model specification. Running instances then have to be either invalidated, restarted, or migrated into the new specification. Unlike the other types of flexibility, flexibility by change is a property of the modelling technique and not just of the specified process. Techniques that support this form of flexibility are ideal for dynamic processes.

2.5 Disease Surveillance

2.5.1 Definition

Disease surveillance as defined by the Centre for Disease Control and Prevention (CDC) [17] is the ongoing, systematic collection, analysis, interpretation, and dissemination of data about a health-related event for use in public health action to reduce morbidity and mortality and to improve health [85]. One of the objectives of disease surveillance is to detect outbreaks, facilitate decision making, and take necessary actions to curb down the spread of the disease [134, 7, 65]. The earlier the detection, the more effective the control and prevention measures will be within the window of opportunity beyond which the epidemic goes out of control [29].

Syndromic surveillance or early warning disease surveillance is the real-time (or near real-time) collection, analysis, interpretation, and dissemination of health-related data to enable the early identification of the impact (or absence of impact) of potential human or veterinary public health threats that require effective public health action [110]. Syndromic surveillance is based not on the laboratory-confirmed diagnosis of a disease but on non-specific health indicators including clinical signs, symptoms as well as proxy measures (eg, absenteeism, drug sales, animal production collapse) that constitute a provisional diagnosis (or “syndrome”). The data is usually collected for purposes other than surveillance and, where possible, are automatically generated so as not to impose an additional burden on the data providers. This surveillance tends to be non-specific yet sensitive and rapid, and can augment and complement the information provided by traditional test-based surveillance systems [110].

Several such systems have been developed all over the world [68, 84, 56, 19, 23, 82] supported by the apparition of new tools in public health (ICT, telecommunication networks, new outbreak detection algorithms, rapid diagnostic tests, etc.), and the availability of data, the proliferation

of new data sources and storage facilities.

The overall objective of syndromic surveillance is to rapidly propose counter conservative measures to hinder the advancement of the epidemic and to initiate aetiological studies to identify the origin of the epidemic. Once the aetiology is established, definitive counter measures can be proposed.

2.5.2 Human Centeredness

Disease (syndromic) surveillance is a multidisciplinary, multi organizational, time critical, decision oriented, data intensive, dependent on information technology, knowledge intensive, and complex process [123]. Standards and regulations [128, 39, 129] prescribe activities, their ordering and how actors interact with them. However, in syndromic surveillance, investigating an outbreak alarm or managing an epidemic is a highly cognitive process [108, 91, 89]. Actors decide on what to do and how to proceed on-the-fly based on their expertise, the available data and resources, and other contextual variables.

Syndromic surveillance is thus an expert medical activity whose objective is to manage a complex situation under temporal constraints and characterised by intense interactions between the different components, uncertainty, and risks [109, 107, 89, 79].

Evaluating a situation, following-up its evolution, and taking timely decisions are characteristics of most professional activities that require considerably high levels of expertise [132]. Such activities are concerned mainly with the supervision and control of dynamic processes. Dynamic because of the continuously changing nature and structure of tasks independently of the actors, and because of the need to make decisions dynamically [13, 50]. In [91], disease surveillance is described as a Joint Cognitive System: one characterised by the association of human and artificial agents working together on coordinated tasks with a shared goal through information and knowledge sharing.

2.5.2.1 Situational Diagnosis

Outbreak alarms identified in syndromic surveillance systems need to be investigated. An outbreak alarm is a statistical aberration of an epidemiological signal obtained when an observed surveillance indicator goes beyond a defined threshold in some outbreak detection algorithm [108].

From an actor's viewpoint, an outbreak alarm has two objectives: raise awareness amongst disease surveillance stakeholders and initiate conservative counter measures [108, 107] while waiting for the alarm to be confirmed (a confirmed alarm is called an outbreak alert).

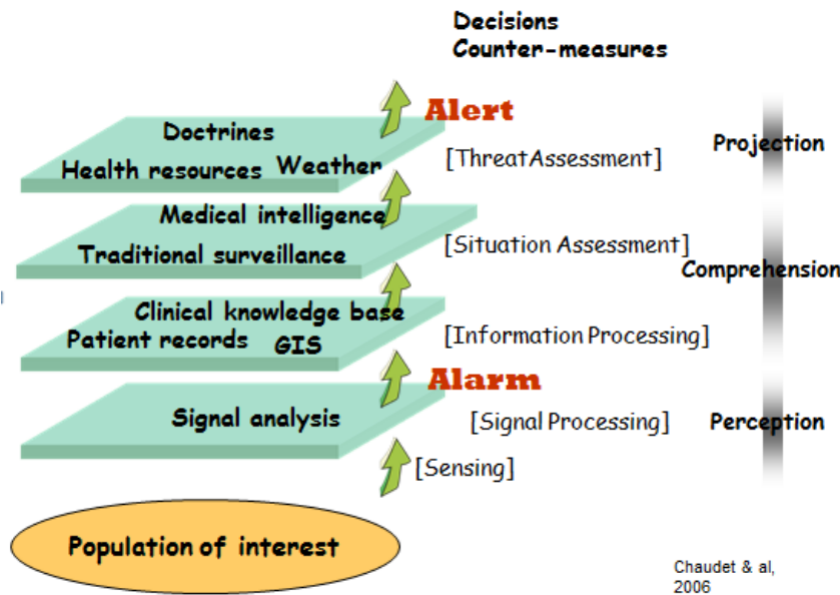


Figure 2.2 – Schematic overview of the model-based architecture for situational diagnosis [18]

The human process between an outbreak alarm and an outbreak alert is called situational diagnosis [18]. This process guides a human expert through the comprehension of the current situation, the projection of future threats, and extension from the projection of alarms to the release of confirmed alerts (Figure 2.2). During situational diagnosis, the experts make use of other data sources (surveys, health information systems, patient records, geographic information systems, weather and climate information, etc), run an indefinite number of complementary analyses, and use related biological diagnosis data to confirm or discard outbreak alarms. One important step before implementing counter measures in the case of an alert is assessment the risk and impact of the epidemic.

2.5.3 Data Drivenness

The complex, dynamic, and cognitive nature of disease surveillance entails that human experts describe what to do on a case by case basis. In other words, each activity on the field is always contextualized based on the available data, their form or structure, and even on some condition expressed on a combination of values.

Several research works [68, 31, 133, 47, 48, 58, 99] have classified disease surveillance and several other health processes as highly data centric and knowledge intensive. Data in disease surveillance is used not only to compute values for relevant indicators and provide more insight to different events and situations that may arise, but also to (manually or automatically) circumscribe and reduce the set of doable tasks and activities to a strict minimum, easily manipulable by a human expert.

2.5.4 Timeliness and Uncertainty

The aim of monitoring diseases is to be able to detect outbreaks early enough and to respond as early as possible within a predefined window of opportunity beyond which the effects of the outbreak might be uncontrollable [7, 68, 134]. Figure 2.3 visually expresses the importance in terms of number of cases (the disease burden) of detection outbreak early enough.

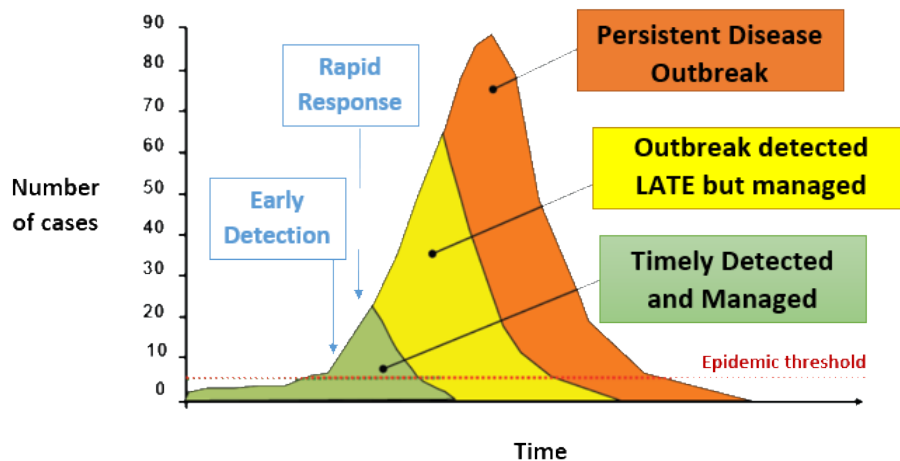


Figure 2.3 – Importance in terms of disease burden of timely detection of disease outbreaks [107]

The process is also characterised by users making time-bound decisions under the pressure induced by disease outbreaks [107, 109]. At the onset, these decisions are made based on incomplete or unrelated data and information, and as time goes on, coupled with the proliferation of surveillance systems, new data sources, users find themselves overwhelmed by the quantity of information that needs to be processed to make decisions. Vis-a-vis the user, this creates uncertainty in decision making.

In a recent publication, Texier et al. [109] supported by previous work in [47] hypothesize that in outbreak situations, the human actors' subjective experiences of uncertain events is a major element of cognitive activities performed to manage the situation. The study suggests that in disease surveillance, uncertainty in decision making has three major causes: insufficient or unavailable information, inadequate understanding due to ambiguities in available information, and conflicts resulting from alternative approaches with equally attractive outcomes.

A process modelling technique applicable in disease surveillance should thus not only provide users with as much information as possible, but also break down the information for easy sorting and filtering to enhance the cognitive capabilities of the human actor and ergonomics of the system and hence improve efficiency [107].

Conclusion

In Part I, we introduced the context and defined the objectives of our work. We then went ahead to present a synthesis of the underlying concepts substantiated with a state of the art of related works with respect to dynamic (collaborative) process modelling.

In conclusion, though several works have addressed dynamic business processes, most have focused on augmenting existing (process centric) techniques to support the flexibility requirements of dynamic processes. Only recently have researchers started working on formal specifications for dynamic case management. Still these recent efforts though geared towards more declarative and data driven approaches leave out one key perspective of dynamic collaborative processes: the user perspective. In this work, we will propose a holistic modelling technique that naturally includes a process, data, and user perspective.

We end this part with a characterization of the disease surveillance process as dynamic, collaborative, human-centered, and data driven. In most of our work we use snippets of disease surveillance scenarios to illustrate different aspects of our model.

In Part II, we present the Guarded Attribute Grammar based dynamic process modelling technique, its properties, and how it is used in Active-Workspaces to structure the work of individual users and ensure effective collaboration between users.

Part II

GUARDED ATTRIBUTE GRAMMARS

Introduction

Case-management usually consists in assembling relevant information by calling *tasks*, which may in turn call subtasks. Case elicitation needs not be implemented as a sequence of successive calls to subtasks, and several subtasks can be performed in parallel. To allow as much concurrency as possible in the execution of tasks, we favour a *declarative* approach where task dependencies are specified without imposing a particular execution order.

Attribute grammars [60, 86] are particularly adapted to that purpose. The model proposed in this work is a variant of attribute grammars, called *Guarded Attributed Grammar* (GAG). We use a notation reminiscent of unification grammars and inspired by the work of Deransart and Maluszynski [26] relating attribute grammars with definite clause programs.

A production of a grammar is, as usual, described by a left-hand side, indicating a non-terminal to expand, and a right-hand side, describing how to expand this non-terminal. We furthermore interpret a production of the grammar as a way to decompose a task (the symbol in the left-hand side of the production) into sub-tasks associated with the symbols in its right-hand side. The semantic rules basically serve as a glue between the task and its sub-tasks. They indicate how the outputs (synthesized attributes) are obtained from the inputs (inherited attributes).

In this declarative model, the lifecycle of artifacts is left implicit. Cases under execution can be seen as incomplete structured documents, i.e., trees with *open nodes*, also called *artifacts*, corresponding to parts of the document that remain to be completed. Each open node is attached a so-called *form* interpreted as a task. A form consists of a task name together with some inherited attributes (data resulting from previous executions) and some synthesized attributes. The latter are variables subscribing to the values that will emerge from task execution.

Productions are *guarded* by patterns occurring at the inherited positions of the left-hand side symbol. Thus, a production is enabled at an open node if the patterns match with the corresponding attribute values as given in the form. The evolution of the artifact thus depends both on previously computed data (stating which production is enabled) and the stakeholder's decisions (choosing a particular production amongst those which are enabled at a given moment and inputting associated data). Thus, GAGs are both *data-driven* and *user-centric*.

Data manipulated in guarded attributed grammars are of two kinds. First, the tasks communicate using *forms* which are temporary information used for communication purposes only, essentially for requesting values. Second, *artifacts* are structured documents that record the history of cases (log of the system).

An artifact grows monotonically –we never erase information. Moreover, every part of the artifact is edited by a unique stakeholder –the owner of the corresponding nodes– hence avoiding edition conflicts. These properties are instrumental to obtain a simple and robust model that can easily be implemented on a distributed asynchronous architecture.

The remainder of this part is organised thus: in Chapter 3 we present the idea of our model, its formal specification, its basic properties, and some specification examples. In Chapter 4, we exhibit certain key properties of the model that are indispensable for dynamic collaborative process modelling, and we also verify the Soundness property of Guarded Attribute Grammars.

Chapter 3

Guarded Attribute Grammars - GAG

This chapter is devoted to a presentation of the model of guarded attribute grammars. We start with an informal presentation that shows how rewriting rules can be used to enforce hierarchical task decomposition, then we introduce guarded attribute grammars and use them to formally structure the notions of task, business-rule, and data, and to show how task resolution is flexibly achieved. The chapter ends with basic examples used to illustrate some of the fundamental characteristics of the model.

3.1 A Grammatical Approach to Task Decomposition

Task analysis as a subject has been broadly studied in social and cognitive sciences [88] and in human-computer-interaction (HCI) modelling and design [21, 95]. One widely studied aspect is *task-decomposition* which employs the hierarchical-task-analysis (HTA)[102] technique to describe user actions, structure them into hierarchies, and prescribe an ordering between them. It also uses Knowledge-Based-Analysis [126] techniques to identify and describe the knowledge requirements of tasks. Task analysis is beyond the scope of our work. Moreover, the output of task analysis is simply a detailed and comprehensive description of tasks which still has to be designed or scripted in some enactment system. Our focus is on designing such a system.

Hierarchical task decomposition can be viewed as a rewriting system in which each task is reduced to its subtasks. A task is thus perceived as a problem to be solved and the rewriting system describes how it can be solved. For instance, let us consider the process definition for a doctor/patient encounter – a medical *consultation*. We can imagine that to consult a patient, the doctor will have to clinically examine the patient, administer some initial care, and if need be declare the patient as a suspect case of some disease under surveillance. It is clear from this description how consultation is done. It can thus be expressed using the following rewriting

rule:

$$\text{consultation} \rightarrow \begin{array}{l} \text{clinical_assessment} \\ \text{initial_care} \\ \text{case_declaration} \end{array}$$

We can further imagine that in order to clinically examine the patient, the doctor needs to interview the patient to obtain some descriptive information (name, age, origin, etc.), his medical history, the current complaint, take measures of some medical indicators (temperature, weight, etc.) based on the previous information, etc. Again, this can be expressed by a rewriting rule:

$$\text{clinical_assessment} \rightarrow \begin{array}{l} \text{interview_patient} \\ \text{obtain_medical_history} \\ \text{take_measures} \end{array}$$

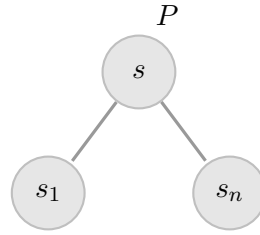
The task decomposition process is continuous and iterative [102]. It is not clearly defined when to stop task refinement but most researchers in the domain clearly agree on stopping either when it can be judged that failure to perform the subtask(s) is inconsequential, or when the subtasks are clear enough (both to the analyst and the subject expert) that they can be reduced to atomic actions such as simple data production actions [102].

In the illustrations above, we have only imagined some possible scenarios but in practice there are numerous ways a patient/doctor encounter might unfold. Each of the possible unfoldings can be expressed as a different rewriting rule. They correspond therefore to a decision that will have to be made by the user, in this case the doctor. For instance, the following two rewriting rules show that declaring a suspect case can unfold in two different ways. The clinician has to decide whether the case under investigation has to be declared to the Disease Surveillance Center or not.

$$\begin{array}{l} \text{suspect_case} : \text{case_declaration} \rightarrow \text{follow_up} \\ \text{benign_case} : \text{case_declaration} \rightarrow \end{array}$$

If the case is reported as a suspect case, then the clinician will have to follow up the case according to further requests of the biologist or of the epidemiologist. On the contrary, if the clinician has described the case as benign, it is closed with no follow up actions. More generally the tasks on the right-hand side of each production represent what remains to be done to resolve the task on the left-hand side in case this rewriting rule is chosen.

In a first approximation, a rewriting rule can be modelled by a *production* $P : s_0 \rightarrow s_1 \cdots s_n$ of a context free grammar expressing that task s_0 can be reduced to subtasks s_1 to s_n . We call such productions *business rules*, and the s_i $i \in [0, n]$, represent the *sorts* (names) of the task and its subtasks and corresponds to non-terminal symbols of the context free grammar. Visually, a business rule which reduces a task of sort s to n tasks of respective sorts s_1, \dots, s_n is represented thus:

Figure 3.1 – Business rule for a task of sort s

Continually refining the subtasks generates a grammar which inherently contains all possible unfoldings of some modelled process. Plain context-free grammars however are not sufficient to model the interactions and data exchanged between the various tasks associated with open nodes. For that purpose, we use attribute grammars [60] which attach additional information, *attributes*, to sorts of the context-free grammar. We will see subsequently how attributes attached to sorts of business rules account for semantic dependencies and data exchanges between a task and its subtasks and between subtasks.

Each sort $s \in S$ comes equipped with a set of *inherited* attributes $Inh(s)$ and a set of *synthesized* attributes $Syn(s)$. We interpret inherited attributes as information needed by the subtasks to compute values for the synthesized attributes and passed down by the parent task. Unlike with traditional attribute grammars, the semantic domain and hence the semantic equations of the grammar are implicitly contained in the right-hand sides of the grammar productions. In other words, synthesized attributes can be viewed as work delegated to subtasks and whose results (values) are obtained by the execution of one or more subtasks. We say that the synthesized attributes of the parent task **subscribe** to values produced by the subtasks.

3.1.1 Modelling Attributes

Data in (guarded) attribute grammars is modelled as attributes associated with symbols of the grammar. Unlike with classical attribute grammars, attributes in guarded attribute grammars are defined as *terms* over a ranked alphabet (a signature) Σ .

Recall 3.1.1 (On Signatures and Terms). Given a set A of symbols (also called constructors), we identify a multi-sorted **signature** Σ on the symbols in A and the set S of sorts as

$$a \in \Sigma \quad a :: (s_1, \dots, s_n) \rightarrow s \quad s_i, s \in S$$

We say that a has arity $n \geq 0$, and sort s . When $S = \{\bullet\}$, i.e. Σ is uni-sorted, we talk of **terms**. Terms may contain undefined parts – the contexts – identified by variables. Let X be the set of variables in a term, then we represent the set of terms on Σ and a countably infinite

set of variables X by the set of trees $T(\Sigma, X)$ defined as follows:

$$\frac{t_1 :: s_1 \quad t_2 :: s_2 \quad \cdots \quad t_n :: s_n \quad a :: s_1 \cdots s_n \rightarrow s}{a(t_1, \dots, t_n) :: s}$$

Definition 3.1. *The set of **terms** over Σ and the set X of variables, denoted $T(\Sigma, X)$ is defined inductively as:*

- i. $x \in T(\Sigma), \forall x \in X$
- ii. *If a is an n -ary symbol ($n \geq 0$) and $t_1, \dots, t_n \in T(\Sigma)$, then $a(t_1, \dots, t_n) \in T(\Sigma)$*

The terms t_i are called the *arguments* of the term $a(t_1, \dots, t_n)$ and the symbol a , the *head symbol* or *root* of the term.

A term is said to be *closed* if it contains no variables. We note $T_0(\Sigma)$ the set of closed terms, also called ground terms.

By $Var(t)$, we denote the set of variables that occur in a term t . So t is closed if $Var(t) = \emptyset$. If $V \subset X$, we can write $T(\Sigma, V)$ to denote the set of terms t with $Var(t) \subseteq V$.

End of Recall 3.1.1

By **data**, we mean any arbitrarily typed piece of information that is needed to complete the execution of a task or that is produced as the result of the execution of a task. For simplicity, we don't assign explicit types to data values but instead consider a **record-syntax representation** adapted for complex data values. Terms have been used in the past to describe data records. The function symbols represent **data constructors** and the subterms represent the **fields** of the record, which are also terms. For example, a *patient* in the scenario above may be characterized by the fields *name*, *dob* (date of birth), and *gender*. We can also imagine that the date of birth is described by the fields *day*, *month*, and *year*. Using the notation for terms, the patient data object is written as:

$$\underline{patient} (\text{name}, \underline{dob} (\text{day}, \text{month}, \text{year}), \text{gender}).$$

As a convention, value constructors are written underlined. Thus, in the above example, patient and dob are data constructors while *name*, *day*, *month*, *year* and *gender* are variables. Let t_a be a term representation for an attribute a . We denote by $Var(t_a)$, the set of variables in the term t_a . Variables in terms represent holes or place-holders which identify the parts of the term that remain to be filled for it to be *closed*. A closed term therefore contains only data constructors (including constant value constructors that carry some meaning in the context in which they are used), or actual data values. For example, a closed instance of the patient term above could be,

$$\underline{patient} ("Peter Pence", \underline{dob} (12, 05, 1975), \underline{male})$$

where the `name` is replaced by a string, the date of birth is instantiated with integer values for the `day`, `month`, and `year`, and the `gender` variable is instantiated with the constant data constructor `male` for the masculine gender.

This form of data modelling is a little subtler than with traditional attribute grammars. While in the latter, semantic rules are defined for each defined-occurrence of attributes $a \in Occ_{def}$, here, semantic rules are defined for all variables occurring in t_a , $a \in Occ_{def}$. Thus, each attribute may have several semantic rules for each of its variables. We therefore talk of semantic rules for variables $x \in Var(t_a)$.

Let $p : s_0 \rightarrow s_1 \dots, s_n$ be a production of a (guarded) attribute grammar G , and $s \in \{s_0, \dots, s_n\}$ be an occurrence of a grammar symbol in p . The sets $inh(s)$ and $syn(s)$ respectively represent the inherited and synthesized attributes associated with s . We define the sets of variable occurrences in attributes of grammar symbols in p as follows:

$$\begin{aligned}
 Var_{inh}(s) &= \{x \in Var(t_a) \mid a \in inh(s)\} \\
 Var_{syn}(s) &= syn(s) \\
 Var(s) &= Var_{inh}(s) \cup Var_{syn}(s) \\
 Var_{inh}(p) &= Var_{inh}(s_0) \\
 Var_{syn}(p) &= Var_{syn}(s_0) \\
 Var(p) &= \bigcup_{\text{for every } s \text{ appearing in } p} Var(s)
 \end{aligned}$$

In like manner, we define the input and output occurrences of variables in p . A variable is said to be an input occurrence if it occurs in an inherited position of the left-hand side of p or in a synthesized position of a symbol in the right-hand side of p . Similarly, a variable is said to be an output occurrence if it occurs in a synthesized position of the left-hand side of p or in an inherited position of a symbol in the right-hand side of p . More precisely,

$$\begin{aligned}
Var_{inp}(p) &= \left(\bigcup_{s \in lhs(p)} Var_{inh}(s) \right) \cup \left(\bigcup_{s \in rhs(p)} Var_{syn}(s) \right) \\
Var_{out}(p) &= \left(\bigcup_{s \in lhs(p)} Var_{syn}(s) \right) \cup \left(\bigcup_{s \in rhs(p)} Var_{inh}(s) \right)
\end{aligned}$$

Variables in $Var_{inp}(p)$ are also called *defined occurrences*, and variables in $Var_{out}(p)$ are also called *used occurrences*. Also, variables in $Var_{inp}(p)$ are pairwise disjoint. In other words, a variable has a unique defined occurrence but can have several used occurrences.

Data plays two important roles in guarded attribute grammars.

- Data gives meaning to the operational semantics of the modelled process: It fuels the evaluation of semantic rules and when data values are computed, they aid to give contextual meaning to the modelled process.
- Terms, also called **patterns** serve as *guards* that can be used to filter data objects. When placed in inherited positions of the left-hand side of a production, patterns are used to specify the form of the values expected by the production. Thus, the production is only applicable if the input values *match* the form of its inherited attributes. Such terms are called *guards* and we say that the production p is *guarded*, hence the name **guarded attribute grammars**.

For instance, suppose a production has two inherited attributes $a_1 = \underline{patient}(\underline{name}, \underline{male})$ and $a_2 = \underline{dob}(\underline{day}, \underline{month}, 1989)$. These express that p is only applicable if it receives as input two attributes with data constructors $\underline{patient}$ and \underline{dob} , and again the patient must be masculine (\underline{male}) born in 1989. Variables \underline{name} , \underline{day} , and \underline{month} are not filtered and simply substituted by the corresponding values.

3.1.2 Modelling Tasks

When tasks are decomposed into task-hierarchies, to resolve a task it suffices to resolve its sub-tasks. We model the decomposition of a task as a production of a guarded attribute grammar. Each symbol (sort) of such a production is a *form* of the corresponding sort.

Definition 3.2 (Forms). A **form** of sort s is an expression $F = s(t_1, \dots, t_n)\langle u_1, \dots, u_m \rangle$ where t_1, \dots, t_n (respectively u_1, \dots, u_m) are terms over a ranked alphabet and a set of variables $var(F)$. Terms t_1, \dots, t_n give the values of the **inherited attributes** and u_1, \dots, u_m the values of the **synthesized attributes** attached to form F .

We call *business rules* or simply *rules*, productions whose sorts are replaced by forms of the corresponding sorts. More precisely, a production $p : s_0 \rightarrow s_1, \dots, s_k$ is rewritten into the following business rule

$$\begin{aligned} s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle &\rightarrow s_1(t_1^{(1)}, \dots, t_{n_1}^{(1)})\langle y_1^{(1)}, \dots, y_{m_1}^{(1)} \rangle \\ &\quad \vdots \\ &\quad s_k(t_1^{(k)}, \dots, t_{n_k}^{(k)})\langle y_1^{(k)}, \dots, y_{m_k}^{(k)} \rangle \end{aligned}$$

where the p_i , the u_j , and the $t_j^{(\ell)}$ are terms and the $y_j^{(\ell)}$ are variables. The forms in the right-hand side of a rule are *tasks* given by forms

$$F = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$$

where the synthesized positions are (distinct) variables y_1, \dots, y_m –i.e., they are not instantiated. The rationale is that we invoke a task by filling in the inherited positions of the form –the entries– and by indicating the variables that expect to receive the results returned during task execution –the *subscriptions*.

If $s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ is a task and $s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ is the left-hand side of a business rule R , we resolve s_0 with R as follows

- check that for $1 \leq i \leq n$, the patterns p_i match the data d_i and create a substitution used to initialize (certain) variables in input positions of tasks in the right-hand side of R ,
- ensure that there is promise of a value for each of the variables y_j $1 \leq j \leq m$, that is, a value subscription is created for every y_j .

When these happen, the task s_0 is replaced by new created tasks s_1, \dots, s_k , corresponding to (instantiated) subtasks in the right-hand side of R . The values of u_j 's are then (eventually) returned to the corresponding variables y_j 's that subscribed to these values. For instance, applying rule (see Fig. 3.2)

$$R : s_0(a(x_1, x_2))\langle b(y'_1), y'_2 \rangle \rightarrow s_1(c(x_1))\langle y'_1 \rangle \quad s_2(x_2, y'_1)\langle y'_2 \rangle$$

to resolve a task $s_0(a(t_1, t_2))\langle y_1, y_2 \rangle$ gives rise to the substitution $x_1 = t_1$ and $x_2 = t_2$. The two newly-created tasks correspond respectively to the subtasks $s_1(c(t_1))\langle y'_1 \rangle$ and $s_2(t_2, y'_1)\langle y'_2 \rangle$ and the values $b(y'_1)$ and y'_2 are substituted to the variables y_1 and y_2 respectively.

This formalism puts emphasis on a declarative (logical) decomposition of tasks to avoid over-constrained schedules. Indeed, semantic rules and guards do not impose any ordering on the execution of tasks. The only possible ordering of tasks is that induced by data dependencies between subtasks, and still, an effective sequence relationship between two tasks exists only when the data produced by one is used to guard the execution of the other. Hence such data

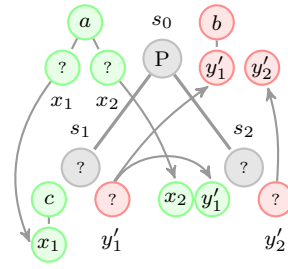


Figure 3.2 – A business rule

values which are simply assigned and not filtered at business rules do not prevent the execution of the corresponding subtasks. In this way, the model allows as much concurrency as possible in process enactment.

As with task analysis, the model can incrementally be designed by observing user’s daily practice and discussing with users. We let users initially develop large chunks of mostly abstract process parts. As their knowledge on the ontology of the system gets enriched, recurrent patterns of activities and their inherent data requirements and exchanges are detected. Users can therefore progressively improve upon the automation of the process with new business processes that either further refine process parts or provide alternative refinements to already refined parts. Thus, an infinite number of business rules can be defined for the same task. These rules share the same sort and may differ in the number and form of their inherited and synthesized attributes, and in the number and type of subtasks in their right-hand sides.

This incremental nature of process design is indispensable in highly dynamic systems. As we show in Section 3.3, the operational semantics of guarded attribute grammars enhance this nature by adding a non-deterministic touch to process enactment and execution.

3.2 Syntax of a Guarded Attribute Grammar

In Section 3.1.2 we described task refinement using business rules into subtasks which can in turn be further refined into new business rules, whose subtasks can be further refined, and so on. This creates a grammar for the resolution of the task – a guarded attribute grammar –.

Attribute grammars, introduced by Donald Knuth in the late sixties [60], have been instrumental in the development of syntax-directed transformations and compiler design. More recently this model has been revived for the specification of structured document’s manipulations mainly in the context of web-based applications. The expression *grammarware* has been coined in [59] to qualify tools for the design and customization of grammars and grammar-dependent software. One such tool is the UUAG system developed by Swierstra and his group. They relied

on purely functional implementations of attribute grammars [55, 97, 9] to build a domain specific languages (DSL) as a set of functional combinators derived from the semantic rules of an attribute grammar [105, 97, 96].

An attribute grammar is obtained from an underlying grammar by associating each sort s with a set $Att(s)$ of *attributes* —which henceforth should exist for each node of the given sort— and by associating each production $P : s \rightarrow s_1 \dots s_n$ with semantic rules describing the functional dependencies between the attributes of a node in its abstract syntax tree labelled P (hence of sort s) and the attributes of its successor nodes —of respective sorts s_1 to s_n . We use a non-standard notation for attribute grammars, inspired from [26, 27]. Let us introduce this notation on an example before proceeding to the formal definition.

In this section, we use the classical example of attribute grammars that computes the flattening of a binary tree, that is, the sequence of leaves read from left to right, to illustrate the syntax we adopted to represent guarded attribute grammars. We then formally define a guarded attribute grammar. We use a non-standard notation for attribute grammars, inspired from [26, 27].

Example 3.2.1 (Flattening of a binary tree).

Our first illustration is the classical example of the attribute grammar that computes the flattening of a binary tree, i.e., the sequence of the leaves read from left to right. The semantic rules are usually presented as shown in Fig. 3.2.1. The sort *bin* of binary trees has two attributes:

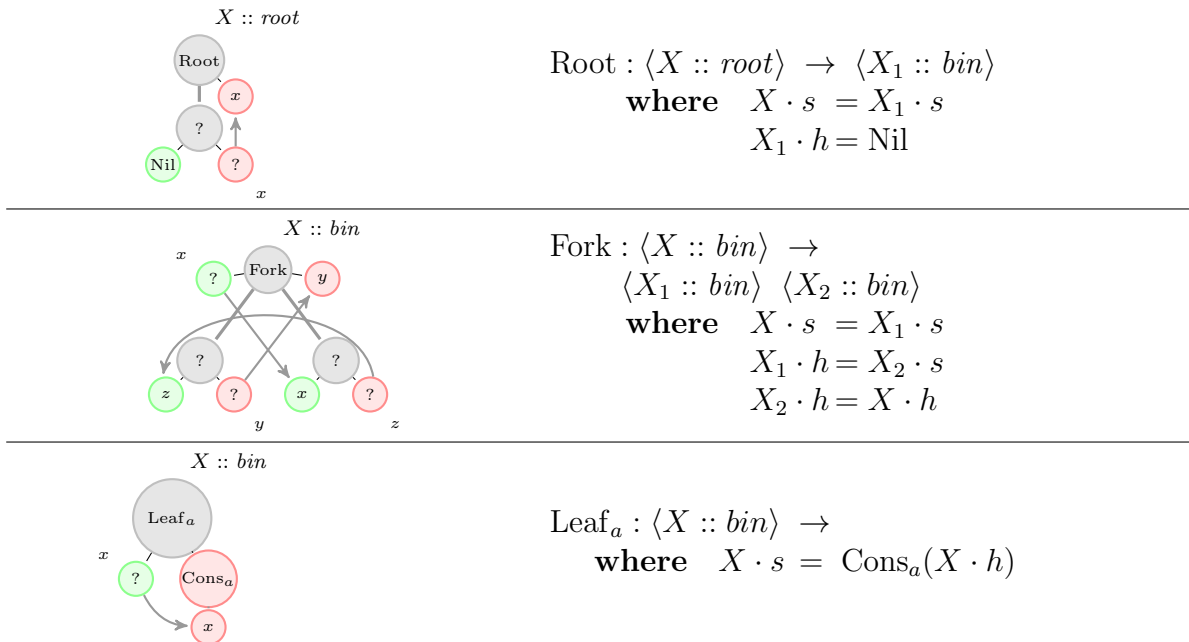


Figure 3.3 – Flattening of a binary tree

The inherited attribute h contains an accumulating parameter and the synthesized attribute s eventually contains the list of leaves of the tree appended to the accumulating parameter. Which we may write as $t \cdot s = flatten(t) ++ t \cdot h$, i.e., $t \cdot s = flat(t, t \cdot h)$ where $flat(t, h) = flatten(t) ++ h$. The semantics rules stem from the identities:

$$\begin{aligned}
\text{flatten}(t) &= \text{flat}(t, \text{Nil}) \\
\text{flat}(\text{Fork}(t_1, t_2), h) &= \text{flat}(t_1, \text{flat}(t_2, h)) \\
\text{flat}(\text{Leaf}_a, h) &= \text{Cons}_a(h)
\end{aligned}$$

We present the semantics rules of Fig. 3.2.1 using the following syntax:

$$\begin{aligned}
\text{Root} &: \quad \text{root}()\langle x \rangle \rightarrow \text{bin}(\text{Nil})\langle x \rangle \\
\text{Fork} &: \quad \text{bin}(x)\langle y \rangle \rightarrow \text{bin}(z)\langle y \rangle \text{bin}(x)\langle z \rangle \\
\text{Leaf}_a &: \quad \text{bin}(x)\langle \text{Cons}_a(x) \rangle \rightarrow
\end{aligned}$$

The *syntactic categories* of the grammar, also called its *sorts*, namely *root* and *bin* are associated with their inherited attributes –given as a list of arguments: (t_1, \dots, t_n) – and their synthesized attributes –the co-arguments: $\langle u_1, \dots, u_m \rangle$. We denote an *input variable* x by $x^?$. It corresponds to a piece of information stemming respectively from the context of the node or from the sub-tree rooted at the corresponding successor node. These variables should be pairwise distinct. Symmetrically, we denote an *output variable* x by $x^!$. It corresponds to values computed by the semantic rules and sent respectively to the context of the node or the sub-tree rooted at the corresponding successor node. Indeed, if we annotate the occurrences of variables with their polarity –input or output– one obtains:

$$\begin{aligned}
\text{Root} &: \quad \text{root}()\langle x^! \rangle \rightarrow \text{bin}(\text{Nil})\langle x^? \rangle \\
\text{Fork} &: \quad \text{bin}(x^?)\langle y^! \rangle \rightarrow \text{bin}(z^!)\langle y^? \rangle \text{bin}(x^!)\langle z^? \rangle \\
\text{Leaf}_a &: \quad \text{bin}(x^?)\langle \text{Cons}_a(x^!) \rangle \rightarrow
\end{aligned}$$

And if we draw an arrow from the (unique) occurrence of $x^?$ to the (various) occurrences of $x^!$ for each variable x to witness the data dependencies then the above rules correspond precisely to the three figures shown on the left-hand side of Table 3.2.1.

Guarded attribute grammars extend the traditional model of attribute grammars by allowing patterns rather than plain variables –as it was the case in the above example– to represent the inherited attributes in the left-hand side of a rule. As mentioned early, patterns allow the semantic rules to process by case analysis based on the shape of some of the inherited attributes, and in this way to handle the interplay between the data –contained in the inherited attributes– and the control –the enabling of rules.

Definition 3.3 (Guarded Attribute Grammars). *Given a set of sorts S with fixed inherited and synthesized attributes, a **guarded attribute grammar** (GAG) is a set of rules $R : F_0 \rightarrow F_1 \cdots F_k$ where the $F_i :: s_i$ are forms. A sort is **used** (respectively **defined**) if it appears in the right-hand side (resp. the left-hand side) of some rule. A guarded attribute grammar G comes with a specific set of sorts $\mathbf{axioms}(G) \subseteq \mathbf{def}(G) \setminus \mathbf{Use}(G)$ –called the **axioms** of G – that are defined and not used. They are interpreted as the provided services. Sorts which are*

used but not defined are interpreted as external services used by the guarded attribute grammar. The values of the inherited attributes of left-hand side F_0 are called the **patterns** of the rule. The values of synthesized attributes in the right-hand side are variables. These occurrences of variables together with the variables occurring in the patterns are called the **input occurrences** of variables. We assume that each variable has **at most one input occurrence**.

Remark 3.2.2. We have assumed in Def. 3.3 that axioms do not appear in the right-hand side of rules. This property will be instrumental to prove that strong-acyclicity –a property that guarantees the safe distribution of the GAG on an asynchronous architecture– can be compositionally verified. Nonetheless a specification that does not satisfy this property can easily be transformed into an equivalent specification that satisfies it: For each axiom s that occurs in the right-hand side of the rule we add a new symbol s' that becomes axioms in the place of s and we add copies of the rules associated with s –containing s in their left-hand side– in which we replace the occurrence of s in the left-hand side by s' . In this way we distinguish s used as a service by the environment of the GAG –role which is now played by s' – from its uses as an internal subtask –role played by s in the transformed GAG. End of Remark 3.2.2

A rule of a GAG specifies the values at output positions –value of a synthesized attribute of s_0 or of an inherited attribute of s_1, \dots, s_n . We refer to these correspondences as the **semantic rules**. More precisely, the inputs are associated with (distinct) variables and the value of each output is given by a term.

A variable can have several occurrences. First it may appear (once) as an input and it may also appear severally in output values. The corresponding occurrence is respectively said to be in an *input* or in an *output position*. We define the following transformation on rules whose effect is to annotate each occurrence of a variable so that $x^?$ (respectively $x^!$) stands for an occurrence of x in an input position (resp. in an output position).

$$\begin{aligned}
!(F_0 \rightarrow F_1 \cdots F_k) &= ?(F_0) \rightarrow !(F_1) \cdots !(F_k) \\
?(s(t_1, \dots, t_n) \langle u_1, \dots, u_m \rangle) &= s(? (t_1), \dots, ? (t_n)) \langle ! (u_1), \dots, ! (u_m) \rangle \\
!(s(t_1, \dots, t_n) \langle u_1, \dots, u_m \rangle) &= s(! (t_1), \dots, ! (t_n)) \langle ? (u_1), \dots, ? (u_m) \rangle \\
?(c(t_1, \dots, t_n)) &= c(? (t_1), \dots, ? (t_n)) \\
!(c(t_1, \dots, t_n)) &= c(! (t_1), \dots, ! (t_n)) \\
?(x) &= x^? \\
!(x) &= x^!
\end{aligned}$$

The conditions stated in Definition 3.3 say that in the labelled version of a rule each variable occurs at most once in an input position, i.e., that $\{?(F_0), !(F_1), \dots, !(F_k)\}$ is an admissible labeling of the set of forms in rule R according to the following definition.

Definition 3.4 (Link Graph). A labelling in $\{?, !\}$ of the variables $\text{var}(\mathcal{F})$ of a set of forms \mathcal{F} is **admissible** if the labelled version of a form $F \in \mathcal{F}$ is given by either $!F$ or $?F$ and each variable has at most one occurrence labelled with $?$. The occurrence $x^?$ identifies the place where the value of variable x is defined and the occurrences of $x^!$ identify the places where this value is used. The **link graph** associated with an admissible labelling of a set of forms \mathcal{F} is the directed graph whose vertices are the occurrences of variables with an arc from v_1 to v_2 if these vertices are occurrences of a same variable x , labelled $?$ in v_1 and $!$ in v_2 . This arc, depicted as follows,



means that the value produced in the **source vertex** v_1 should be forwarded to the **target vertex** v_2 . Such an arc is called a **data link**.

Definition 3.5 (Underlying Grammar). The **underlying grammar** of a guarded attribute grammar G is the context-free grammar $\mathcal{U}(G) = (N, T, A, \mathcal{P})$ where

- the non-terminal symbols $s \in N$ are the defined sorts,
- $T = S \setminus N$ is the set of terminal symbols –the external services–,
- $A = \mathbf{axioms}(G)$ is the set of axioms of the guarded attribute grammar, and
- the set of productions \mathcal{P} is made of the underlying productions $\mathcal{U}(R) : s_0 \rightarrow s_1 \cdots s_k$ of rules $R : F_0 \rightarrow F_1 \cdots F_k$ with $F_i :: s_i$.

A guarded attribute grammar is said to be **autonomous** when its underlying grammar contains no terminal symbols.

Intuitively an autonomous guarded attribute grammar represents a standalone application: It corresponds to the description of particular services, associated with the axioms, whose realizations do not rely on external services.

3.3 Behaviour of a Guarded Attribute Grammar

Executing a guarded attribute grammar specification of a process is analogous to constructing an abstract syntax trees for the underlying attribute grammar. In the latter case, these trees are usually produced by some parsing algorithm during an earlier step. The semantic rules are then used to decorate the nodes of the input tree with attributes and attribute values. In our setting, the generation of the tree and its evaluation using the semantic rules are intertwined, that is, each rule is decorated with its semantic-rules the moment it is plugged into the abstract syntax tree. The abstract syntax tree is therefore viewed as partially constructed with internal

nodes (closed nodes) representing tasks for which a resolution method has been assigned, and leaf nodes (open nodes) representing pending tasks. We call this tree under construction, an artifact.

An artifact is thus an (incomplete) abstract syntax tree that registers all work that has been done till a certain instant and brings to the limelight what remains to be done. A closed node in an artifact is labelled by the rule that was used to create it and an open node is associated with a form (a task) that will contain all the needed information for its eventual refinement. The information attached to an open node consists of the sort of the node, the current value of its inherited attributes, and the set of applicable business rules obtained after filtering the guards. The synthesized attributes of an open node are undefined and are thus associated with variables.

We describe and illustrate the operational semantics of a guarded attribute grammar using the following three concepts: configuration (snapshot) and atomic (business) step. A configuration allows to characterize a snapshot of the enacted artifact at some point in time, atomic steps sanction the move from a configuration to the next, and data-update steps are used to update data values for variables in inherited positions of the business rule.

3.3.1 Configuration of a Guarded Attribute Grammar

We consider in this section, only autonomous guarded attribute grammars. The definitions however can be easily generalized to all guarded attribute grammars.

Definition 3.6 (Configuration). *A **configuration** Γ of an autonomous guarded attribute grammar with underlying grammar G is an S -sorted set of nodes $X \in \text{nodes}(\Gamma)$ each of which is associated with a defining equation in one of the following forms where $\text{Var}(\Gamma)$ is a set of variables associated with Γ :*

Closed node: $X = R(X_1, \dots, X_k)$ where $X :: s$, and $X_i :: s_i$ for $1 \leq i \leq k$, and $\mathcal{U}(R) : s \rightarrow s_1 \dots s_k$ is the underlying production of rule R . Rule R is the **label** of node X and nodes X_1 to X_n are its **successor nodes**.

Open node: $X = s(t_1, \dots, t_n)\langle x_1, \dots, x_m \rangle$ where X is of sort s and t_1, \dots, t_n are terms with variables in $\text{Var}(\Gamma)$ that represent the values of the inherited attributes of X , and x_1, \dots, x_m are variables in $\text{Var}(\Gamma)$ associated with its synthesized attributes. – We denote by $\text{Var}_{inh}(\Gamma)$ and $\text{Var}_{syn}(\Gamma)$, the sets of inherited and synthesized variables of Γ respectively –.

Each variable in $\text{var}(\Gamma)$ occurs at most once in a synthesized position. Otherwise stated $!\Gamma = \{!F \mid F \in \Gamma\}$ is an admissible labelling of the set of forms occurring in Γ . A node is called a **root node** when its sort is an axiom. Each node is the successor of a unique node, called

its **predecessor**, except for the root nodes that are the successor of no other nodes. Hence a configuration is a set of trees –abstract-syntax trees of the underlying grammar– which we call the **artifacts** of the configuration. Each axiom is associated with a **map** made of the artifacts of the corresponding sort. A map thus collects the artifacts corresponding to a specific service of the GAG.

In order to specify the effect of applying a rule at a given node of a configuration (an Atomic Step, Definition 3.10) we first recall some notions about substitutions.

Recall 3.3.1 (on Substitutions). A substitution defines for each variable in a set of variables, a value that should be substituted for all occurrences of the variable.

Definition 3.7 (Substitution). A substitution σ on a set of variables $X = \{x_1, \dots, x_k\}$, called the **domain** of σ and denoted $\text{dom}(\sigma)$, is a partial map

$$\sigma : X \mapsto T_{\Sigma}(Y) \quad \text{with} \quad X \cap Y = \emptyset$$

defined by the system of equations

$$\{x_i = t_i \mid 1 \leq i \leq k\}$$

The set $\text{var}(\sigma) = \bigcup_{1 \leq i \leq k} \text{var}(t_i)$ of variables of σ , is disjoint from the domain $\text{dom}(\sigma)$ of σ . We say that $\text{dom}(\sigma)$ is the set of **defined variables** of σ and $\text{var}(\sigma)$ is the set of **used variables** of σ .

Conversely a system of equations $\{x_i = t_i \mid 1 \leq i \leq k\}$ defines a substitution σ with $\sigma(x_i) = t_i$ if it is in **solved form**, i.e., none of the variables x_i appears in some of the terms t_j . In order to transform a system of equations $E = \{x_i = t_i \mid 1 \leq i \leq k\}$ into an equivalent system $\{x_i = t'_j \mid 1 \leq j \leq m\}$ in solved form one can iteratively replace an occurrence of a variable x_i in one of the right-hand side term t_j by its definition t_i until no variable x_i occurs in some t_j .

Definition 3.8 (Applying a substitution). Given a substitution $\sigma : X \mapsto T_{\Sigma}(Y)$ such that $t \in T_{\Sigma}(X) \Rightarrow t\sigma \in T_{\Sigma}(X \cup Y)$, we define the application of a substitution by the following equations:

$$\begin{aligned} x\sigma &= \sigma(x) \quad \text{if } x \in \text{dom}(\sigma) \\ x\sigma &= x \quad \text{if } x \notin \text{dom}(\sigma) \\ a(t_1, \dots, t_n)\sigma &= a(t_1\sigma, \dots, t_n\sigma) \end{aligned}$$

We note $t[t_i/x_i]$, the operation of replacing all occurrences of variables x_i by their values t_i in a term t .

This process terminates when the relation $x_i \succ x_j \Leftrightarrow x_j \in \text{var}(\sigma(x_i))$ is acyclic. One can easily verify that, under this assumption, the resulting system of equation $SF(E) = \{x_i = t'_i \mid 1 \leq i \leq n\}$ in solved form does not depend on the order in which the variables x_i have been eliminated from the right-hand sides. When the above condition is met we say that the set of equations is **acyclic** and that it **defines** the substitution associated with the solved form.

The composition of two substitutions σ, σ' , where $\text{var}(\sigma') \cap \text{dom}(\sigma) = \emptyset$, is denoted by $\sigma\sigma'$ and defined by $\sigma\sigma' = \{x = t\sigma' \mid x = t \in \sigma\}$. Similarly, we let $\Gamma\sigma$ denote the configuration obtained from Γ by replacing the defining equation $X = F$ of each open node X by $X = F\sigma$.

End of Recall 3.3.1

3.3.2 Atomic Step - Applying a Business Rule

The operational semantics of a guarded attribute grammar is described by successive snapshots (configurations) of the enacted system, each snapshot created by the execution of an atomic or business step. An atomic step that sanctions the move from a configuration Γ to a configuration Γ' corresponds to the application of a business rule R at an open node X in Γ . The new configuration Γ' is augmented with a closed node (the previously open node X) and none or several open nodes corresponding to subtasks in the right-hand side of R .

We now define more precisely when a rule is enabled at a given open node of a configuration and the effect executing an atomic step. First, note that variables of a rule are formal parameters whose scopes are limited to the rule. They can injectively be renamed in order to avoid clashes with variable names appearing in the configuration. Therefore, we always assume that the set of variables of a rule R is disjoint from the set of variables of configuration Γ when applying rule R at a node of Γ . As informally stated in Section 3.1.2, a rule R applies at an open node X when its left-hand side $s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ matches with the definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$, that is, the task associated with X in Γ . We use the usual pattern matching algorithm [125] summarized by the following inductive statements

$$\begin{aligned} & \mathbf{match}(c(p'_1, \dots, p'_k), c'(d'_1, \dots, d'_{k'})) \text{ with } c \neq c' \text{ fails} \\ & \mathbf{match}(c(p'_1, \dots, p'_k), c(d'_1, \dots, d'_k)) = \sum_{i=1}^k \mathbf{match}(p'_i, d'_i) \\ & \mathbf{match}(x, d) = \{x = d\} \end{aligned}$$

where the sum $\sigma = \sum_{i=1}^k \sigma_i$ of substitutions σ_i is defined and equal to $\bigcup_{i \in 1..k} \sigma_i$ when all substitutions σ_i are defined and associated with disjoint sets of variables. Note that since no variable occurs twice in the whole set of patterns p_i , the various substitutions $\mathbf{match}(p_i, d_i)$, when defined, are indeed concerned with disjoint sets of variables. Note also that $\mathbf{match}(c(), c()) = \emptyset$.

Definition 3.9. A form $F = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ **matches** with a task invocation $F' = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ –of the same sort– when

1. the patterns p_i match with the data d_i , defining a substitution $\sigma_{in} = \sum_{1 \leq i \leq n} \mathbf{match}(p_i, d_i)$,
2. the set of equations $\{y_j = u_j \sigma_{in} \mid 1 \leq j \leq m\}$ is acyclic and defines a substitution σ_{out} .

The resulting substitution $\sigma = \mathbf{match}(F, F')$ is given by $\sigma = \sigma_{out} \cup \sigma_{in} \sigma_{out}$.

Remark 3.3.2. In most cases variables y_j do not appear in expressions d_i . And when it is the case one has only to check that patterns p_i matches with data d_i –substitution σ_{in} is defined– because then $\sigma_{out} = \{y_j = u_j \sigma_{in} \mid 1 \leq j \leq m\}$ since the latter is already in solved form. Moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ because variables y_j do not appear in expressions $\sigma_{in}(x_i)$.

End of Remark 3.3.2

Definition 3.10 (Atomic Step - Applying a Rule). Let $R = F_0 \rightarrow F_1 \dots F_k$ be a rule, Γ be a configuration, and $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ be an open node in Γ . We assume that R and Γ are defined over disjoint sets of variables. We say that R is **enabled** in X and write $\Gamma[R/X]$, if the left-hand side of R matches with the definition of X . Then applying rule R at node X transforms configuration Γ into Γ' , denoted as $\Gamma[R/X]\Gamma'$, with Γ' defined as follows:

$$\begin{aligned} \Gamma' &= \{X = R(X_1, \dots, X_k)\} \\ &\cup \{X_1 = F_1 \sigma, \dots, X_k = F_k \sigma\} \\ &\cup \{X' = F \sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \end{aligned}$$

where $\sigma = \mathbf{match}(F_0, X)$ and X_1, \dots, X_k are new nodes added to Γ' .

Thus the first effect of applying rule R to an open node X is that X becomes a closed node with label R and new open nodes X_1 to X_k are added to Γ' and associated respectively with the instances of the k forms in the right-hand side of R obtained by applying substitution σ to these forms, that is, $X_i = F_i \sigma$.

The definitions of the other nodes of Γ are updated using substitution σ –or equivalently σ_{out} . This update has no effect on the closed nodes because their defining equations in Γ contain no variable.

We conclude this section with two results justifying Definition 3.10. Namely, Prop. 3.3.3 states that if R is a rule enabled in a node X_0 of a configuration Γ with $\Gamma[R/X_0]\Gamma'$ then Γ' is a configuration: Applying R cannot create a variable with several input occurrences. And Prop. 3.3.5 shows that substitution $\sigma = \mathbf{match}(F_0, X)$ resulting from the matching of the left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ of a rule R with the definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of an open node X is the most general unifier of the set of equations $\{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\}$.

Proposition 3.3.3. *If rule R is enabled in an open node X_0 of a configuration Γ and $\Gamma[R/X_0]\Gamma'$ then Γ' is a configuration.*

Proof. Let $R = F_0 \rightarrow F_1 \dots F_k$ with left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ and $X_0 = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ be the defining equation of X_0 in Γ . Since the values of synthesized attributes in the forms F_1, \dots, F_k are variables (by Definition 3.3) and since these variables are unaffected by substitution σ_{in} the synthesized attribute in the resulting forms $F_j\sigma_{in}$ are variables. The substitutions σ_{in} and σ_{out} substitute terms to the variables x_1, \dots, x_k appearing to the patterns and to the variables y_1, \dots, y_m respectively. Since x_i appears in an input position in R , it can appear only in an output position in the forms $!(F_1), \dots, !(F_k)$ and thus any variable of the term $\sigma_{in}(x_i)$ will appear in an output position in $!(F_i\sigma_{in})$. Similarly, since y_i appears in an input position in the form $!(s(u_1, \dots, u_n)\langle y_1, \dots, y_m \rangle)$, it can only appear in an output position in $!(F)$ for the others forms F of Γ . Consequently any variable of the term $\sigma_{out}(y_i)$ will appear in an output position in $!(F\sigma_{out})$ for any equation $X = F$ in Γ with $X \neq X_0$. It follows that the application of a rule cannot produce new occurrences of a variable in an input position and thus there cannot exist two occurrences $x^?$ of a same variable x in Γ' . $Q.E.D.$

Definition 3.11. *A configuration Γ' is **directly accessible** from Γ , denoted by $\Gamma[\]\Gamma'$, whenever $\Gamma[R/X]\Gamma'$ for some rule R enabled in node X of configuration Γ . Furthermore, a configuration Γ' is **accessible** from configuration Γ when $\Gamma[*]\Gamma'$ where $[\]$ is the reflexive and transitive closure of relation $[\]$.*

Recall that a substitution σ unifies a set of equations E if $t\sigma = t'\sigma$ for every equation $t = t'$ in E . A substitution σ is more general than a substitution σ' , denoted $\sigma \preceq \sigma'$, if $\sigma' = \sigma\sigma''$ for some substitution σ'' . If a system of equations has a some unifier, then it has –up to an bijective renaming of the variables in σ – a *most general unifier*. In particular a set of equations of the form $\{x_i = t_i \mid 1 \leq i \leq n\}$ has a unifier if and only if it is acyclic. In this case, the corresponding solved form is its most general unifier.

Recall 3.3.4 (on Unification). We consider sets $E = E_? \uplus E_=-$ containing equations of two kinds. An equation in $E_?$, denoted as $t \stackrel{?}{=} u$, represents a *unification goal* whose solution is a substitution σ such that $t\sigma = u\sigma$ –substitution σ unifies terms t and u . $E_=-$ contains equations of the form $x = t$ where variable x occurs only there, i.e., we do not have two equations with the same variable in their left-hand side and such a variable cannot either occur in any right-hand side of an equation in $E_=-$. A *solution* to E is any substitution σ whose domain is the set of variables occurring in the right-hand sides of equations in $E_=-$ such that the compound substitution made of σ and the set of equations $\{x = t\sigma \mid x = t \in E_=-\}$ unifies terms t and u for any equation $t \stackrel{?}{=} u$ in $E_?$. Two systems of equations are said to be *equivalent* when they have

the same solutions. A *unification problem* is a set of such equations with $E_{=} = \emptyset$, i.e., it is a set of unification goals. On the contrary E is said to be in *solved form* if $E_{=} = \emptyset$, thus E defines a substitution which, by definition, is the most general solution to E . Solving a unification problem E consists in finding an equivalent system of equations E' in solved form. In that case E' is a *most general unifier* for E .

Martelli and Montanari Unification algorithm [72] proceeds as follows. We pick up non-deterministically one equation in $E_{=}$ and depending on its shape apply the corresponding transformation:

1. $c(t_1, \dots, t_n)c(u_1, \dots, u_n)$: replace it by equations $t_1 \stackrel{?}{=} u_1, \dots, t_n \stackrel{?}{=} u_n$.
2. $c(t_1, \dots, t_n) \stackrel{?}{=} c'(u_1, \dots, u_m)$ with $c \neq c'$: halt with failure.
3. $x \stackrel{?}{=} x$: delete this equation.
4. $t \stackrel{?}{=} x$ where t is not a variable: replace this equation by $x \stackrel{?}{=} t$.
5. $x \stackrel{?}{=} t$ where $x \notin \text{var}(t)$: replace this equation by $x = t$ and substitute x by t in all other equations of E .
6. $x \stackrel{?}{=} t$ where $x \in \text{var}(t)$ and $x \neq t$: halt with failure.

The condition in (5) is the occur check. Thus, the computation fails either if the two terms of an equation cannot be unified because their main constructors are different or because a potential solution of an equation is necessarily an infinite tree due to a recursive statement detected by the occur check. System E' obtained from E by applying one of these rules, denoted as $E \Rightarrow E'$, is clearly equivalent to E . We iterate this transformation as long as we do not encounter a failure and some equation remains in $E_{=}$. It can be proven that all these computations terminate and either the original unification problem E has a solution –a unifier– and every computation terminates –and henceforth produces a solved set equivalent to E describing a most general unifier of E – or E has no unifier and every computation fails. We let

$$\sigma = \mathbf{mgu}(\{t_i = u_i\}_{1 \leq i \leq n}) \quad \text{iff} \quad \left\{ t_i \stackrel{?}{=} u_i \right\}_{1 \leq i \leq n} \Rightarrow^* \sigma$$

End of Recall 3.3.4

Note that Steps (5) and (6) of the unification algorithm are the only rules that can be applied to solve a unification problem of the form $\{y_i \stackrel{?}{=} u_i \mid 1 \leq i \leq n\}$, where the y_i are distinct variables. The most general unifier exists when the occur check always holds, i.e., rule (5) always applies. The computation amounts to iteratively replacing an occurrence of a variable y_i in one of the right-hand side term u_j by its definition u_i until no variable y_i occurs in some u_j , i.e., (see Recall 3.3.1) when this system of equations is acyclic. Hence any acyclic set of equations $\{y_i = u_i \mid 1 \leq i \leq n\}$ defines the substitution $\sigma = \mathbf{mgu}(\{y_i = u_i \mid 1 \leq i \leq n\})$.

Proposition 3.3.5. *If $F_0 = s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$, left-hand side of a rule R , matches with the definition $X = s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of an open node X then substitution $\sigma = \mathbf{match}(F_0, X)$ is the most general unifier of the set of equations*

$$\{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\}$$

.

Proof. If a rule R of left-hand side $s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ is triggered in an open node X_0 with $X_0 = s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ then by Def. 3.10 $\left\{p_i \stackrel{?}{=} d_i\right\}_{1 \leq i \leq n} \cup \left\{y_j \stackrel{?}{=} u_j\right\}_{1 \leq j \leq m} \Rightarrow^* \sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j \sigma_{in}\right\}_{1 \leq j \leq m}$ using only the rules (1) and (5) of the unification algorithm above. Now, by applying iteratively rule (5) one obtains

$$\sigma_{in} \cup \left\{y_j \stackrel{?}{=} u_j \sigma_{in}\right\}_{1 \leq j \leq m} \Rightarrow^* \sigma_{in} \cup \mathbf{mgu} \{y_j = u_j \sigma_{in}\}_{1 \leq j \leq m}$$

when the set of equations $\{y_j = u_j \sigma_{in}\}_{1 \leq j \leq m}$ satisfies the occur check. Then $\sigma_{in} + \sigma_{out} \Rightarrow^* \sigma$ again by using rule (5). *Q.E.D.*

Remark 3.3.6. The converse of Prop. 3.3.5 does not hold. Namely, one shall not deduce from Proposition 3.3.5 that the relation $\Gamma[R/X_0]\Gamma'$ is defined whenever the left-hand side of R ($\text{lhs}(R)$) can be unified with the definition $\text{def}(X_0, \Gamma)$ of X_0 in Γ with

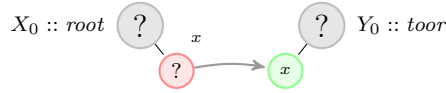
$$\begin{aligned} \Gamma' &= \{X_0 = R(X_1, \dots, X_k)\} \\ &\cup \{X_1 = F_1\sigma, \dots, X_k = F_k\sigma\} \\ &\cup \{X = F\sigma \mid (X = F) \in \Gamma \wedge X \neq X_0\} \end{aligned}$$

where $\sigma = \mathbf{mgu}(\text{lhs}(R), \text{def}(X_0, \Gamma))$, and X_1, \dots, X_k are new nodes added to Γ' . Indeed, when unifying $\text{lhs}(R)$ with $\text{def}(X_0, \Gamma)$ one may generate an equation of the form $x = t$ where x is a variable in an inherited data d_i and t is an instance of a corresponding subterm in the associated pattern p_i . This would correspond to a situation where information is sent to the context of a node through one of its inherited attributes. Stated differently, with this alternative definition some parts of the pattern p_i could actually be used to filter out the incoming data value d_i while some other parts of the same pattern would be used to transfer synthesized information to the context. *End of Remark 3.3.6*

3.4 Examples

In this section we illustrate the behaviour of guarded attribute grammars with two examples. Example 3.4.1 describes an execution of the attribute grammar of Example 3.2.1. The specification in Example 3.2.1 is actually an ordinary attribute grammar because the inherited attributes in the left-hand sides of rules are plain variables. This example shows how data are lazily produced and sent in push mode through attributes. It also illustrates the role of the data links and their dynamic evolutions. Example 3.4.2 illustrates the role of the guards by describing two processes acting as coroutines. The first process sends forth a list of values to the second process and it waits for an acknowledgement for each message before sending the next one.

Example 3.4.1 (Example 3.2.1 continued). Let us consider the attribute grammar of Example 3.2.1 and the initial configuration $\Gamma_0 = \{X_0 = \text{root}()\langle x \rangle, Y_0 = \text{toor}(x)\langle \rangle\}$ shown next



The annotated version $!\Gamma_0 = \{!F \mid F \in \Gamma_0\}$ of configuration Γ_0 is

$$!\Gamma_0 = \{X_0 = \text{root}()\langle x^? \rangle, Y_0 = \text{toor}(x^!)\langle \rangle\}$$

The data link from $x^?$ to $x^!$ says that the list of the leaves of the tree –that will stem from node X_0 – to be synthesized at node X_0 should be forwarded to the inherited attribute of Y_0 .

This tree is not defined in the initial configuration Γ_0 . One can start developing it by applying rule $\text{Root} : \text{root}()\langle u \rangle \rightarrow \text{bin}(\text{Nil})\langle u \rangle$ at node $X_0 :: \text{root}$. Actually the left-hand side $\text{root}()\langle u \rangle$ of rule Root matches with the definition $\text{root}()\langle x \rangle$ of X_0 with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = u\}$. Thus $\Gamma_0[\text{Root}/X_0]\Gamma_1$ where the annotated configuration $!\Gamma_1$ is given in Figure 3.4.

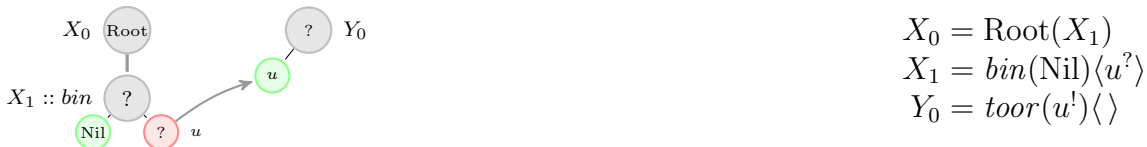
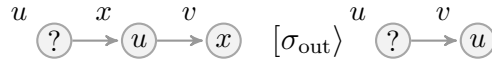


Figure 3.4 – Configuration Γ_1

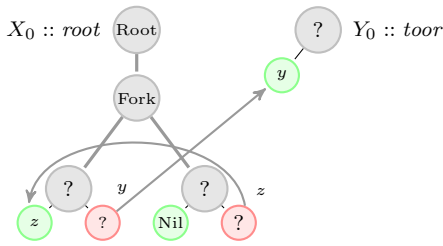
Note that substitution $\sigma_{out} = \{x = u\}$ replaces the data link $(x^?, x^!)$ by a new link $(u^?, u^!)$ with the same target and whose source has been moved from the synthesized attribute of X_0 to the synthesized attribute of X_1 .



The tree may be refined by applying rule

$$\text{Fork} : bin(x)\langle y \rangle \rightarrow bin(z)\langle y \rangle bin(x)\langle z \rangle$$

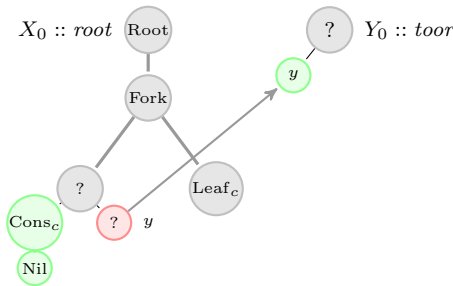
at node $X_1 :: bin$ since its left-hand side $bin(x)\langle y \rangle$ matches with the definition $bin(Nil)\langle u \rangle$ of X_1 with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{u = y\}$. Hence $\Gamma_1[\text{Fork}/X_1]\Gamma_2$ where Γ_2 is given in Figure 3.5.



$$\begin{aligned}
 X_0 &= \text{Root}(X_1) \\
 X_1 &= \text{Fork}(X_{11}, X_{12}) \\
 X_{11} &= bin(z')\langle y' \rangle \\
 X_{12} &= bin(Nil)\langle z' \rangle \\
 Y_0 &= toor(y')\langle \rangle
 \end{aligned}$$

Figure 3.5 – Configuration Γ_2

Rule $\text{Leaf}_c : bin(x)\langle \text{Cons}_c(x) \rangle \rightarrow$ applies at node X_{12} since its left-hand side $bin(x)\langle \text{Cons}_c(x) \rangle$ matches with the definition $bin(Nil)\langle z \rangle$ of X_{12} with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{z = \text{Cons}_c(Nil)\}$. Hence $\Gamma_2[\text{Leaf}_c/X_{12}]\Gamma_3$ where the annotated configuration Γ_3 is given in Figure 3.6.

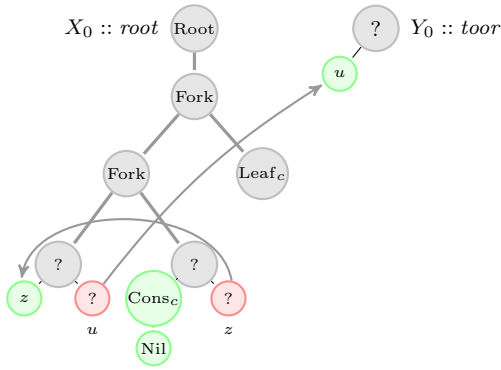


$$\begin{aligned}
 X_0 &= \text{Root}(X_1) \\
 X_1 &= \text{Fork}(X_{11}, X_{12}) \\
 X_{11} &= bin(\text{Cons}_c(Nil))\langle y' \rangle \\
 X_{12} &= \text{Leaf}_c \\
 Y_0 &= toor(y')\langle \rangle
 \end{aligned}$$

Figure 3.6 – Configuration Γ_3

As a result of substitution $\sigma_{out} = \{z = \text{Cons}_c(Nil)\}$ the value $\text{Cons}_c(Nil)$ is transmitted through the link $(z^?, z')$ and this link disappears.

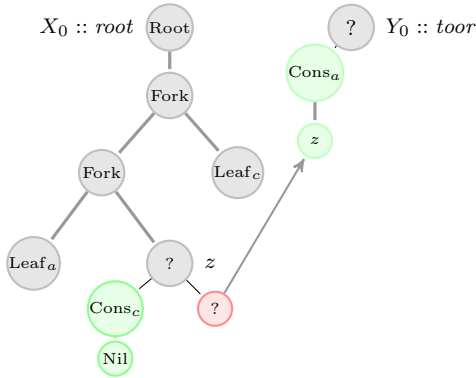
Rule $\text{Fork} : bin(x)\langle u \rangle \rightarrow bin(z)\langle u \rangle bin(x)\langle z \rangle$ may apply at node X_{11} : its left-hand side $bin(x)\langle u \rangle$ matches with the definition $bin(\text{Cons}_c(Nil))\langle y \rangle$ of X_{11} with $\sigma_{in} = \{x = \text{Cons}_c(Nil)\}$ and $\sigma_{out} = \{y = u\}$. Hence $\Gamma_3[\text{Fork}/X_{11}]\Gamma_4$ with configuration Γ_4 given in Figure 3.7.



$$\begin{aligned}
X_0 &= \text{Root}(X_1) \\
X_1 &= \text{Fork}(X_{11}, X_{12}) \\
X_{11} &= \text{Fork}(X_{111}, X_{112}) \\
X_{111} &= \text{bin}(z!) \langle u? \rangle \\
X_{112} &= \text{bin}(\text{Cons}_c(\text{Nil})) \langle z? \rangle \\
X_{12} &= \text{Leaf}_c \\
Y_0 &= \text{toor}(u!) \langle \rangle
\end{aligned}$$

Figure 3.7 – Configuration Γ_4

Rule $\text{Leaf}_a : \text{bin}(x) \langle \text{Cons}_a(x) \rangle \rightarrow$ applies at node X_{111} since its left-hand side $\text{bin}(x) \langle \text{Cons}_a(x) \rangle$ matches with the definition $\text{bin}(z) \langle u \rangle$ of X_{111} with $\sigma_{in} = \{x = z\}$ and $\sigma_{out} = \{u = \text{Cons}_a(z)\}$. Hence $\Gamma_4[\text{Leaf}_a/X_{111}] \Gamma_5$ with configuration $! \Gamma_5$ given in Figure 3.8.



$$\begin{aligned}
X_0 &= \text{Root}(X_1) \\
X_1 &= \text{Fork}(X_{11}, X_{12}) \\
X_{11} &= \text{Fork}(X_{111}, X_{112}) \\
X_{111} &= \text{Leaf}_a \\
X_{112} &= \text{bin}(\text{Cons}_c(\text{Nil})) \langle z? \rangle \\
X_{12} &= \text{Leaf}_c \\
Y_0 &= \text{toor}(\text{Cons}_a(z!)) \langle \rangle
\end{aligned}$$

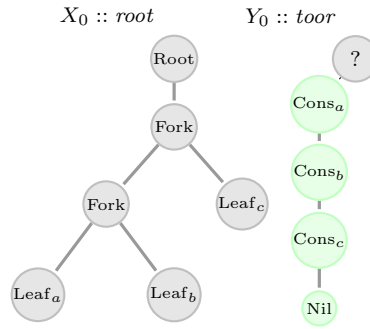
Figure 3.8 – Configuration Γ_5

Using substitution $\sigma_{out} = \{u = \text{Cons}_a(z)\}$ the data $\text{Cons}_a(z)$ is transmitted through the link $(u^?, u^!)$ which, as a result, disappears. A new link $(z^?, z^!)$ is created so that the rest of the list, to be synthesized in node X_{112} can later be forwarded to the inherited attribute of Y_0 .

Finally one can apply rule $\text{Leaf}_b : \text{bin}(x) \langle \text{Cons}_a(x) \rangle \rightarrow$ at node X_{112} since its left-hand side matches with the definition $\text{bin}(\text{Cons}_c(\text{Nil})) \langle z \rangle$ of X_{112} with $\sigma_{in} = \{x = \text{Cons}_c(\text{Nil})\}$ and $\sigma_{out} = \{z = \text{Cons}_b(\text{Cons}_c(\text{Nil}))\}$.

Therefore, $\Gamma_5[\text{Leaf}_b/X_{112}] \Gamma_6$ with configuration $! \Gamma_6$ given in Figure 3.9.

Now the tree rooted at node X_0 is closed –and thus it no longer holds attributes– and the list of its leaves has been entirely forwarded to the inherited attribute of node Y_0 . Note that the recipient node Y_0 could have been refined in parallel with the changes of configurations just



$$\begin{aligned}
 X_0 &= \text{Root}(X_1) \\
 X_1 &= \text{Fork}(X_{11}, X_{12}) \\
 X_{11} &= \text{Fork}(X_{111}, X_{112}) \\
 X_{111} &= \text{Leaf}_a \\
 X_{112} &= \text{Leaf}_b \\
 X_{12} &= \text{Leaf}_c \\
 Y_0 &= \text{toor}(\text{Cons}_a(\text{Cons}_b(\text{Cons}_c(\text{Nil}))))\langle \rangle
 \end{aligned}$$

Figure 3.9 – Configuration Γ_6

described.

End of Example 3.4.1

The above example shows that data links are used to transmit data in push mode from a source vertex v –the input occurrence $x^?$ of a variable x – to some target vertex v' –an output occurrence $x^!$ of the same variable. These links $(x^!, x^?)$ are *transient* in the sense that they disappear as soon as variable x gets defined by the substitution σ_{out} induced by the application of a rule in some open node of the current configuration. If $\sigma_{out}(x)$ is a term t , not reduced to a variable, with variables x_1, \dots, x_k then vertex v' is refined by the term $t[x_i^!/x_i]$ and new vertices $v_i^!$ — associated with these new occurrences of x_i in an output position— are created. The original data link $(x^?, x^!)$ is replaced by all the corresponding instances of $(x_i^?, x_i^!)$. Consequently, a target is replaced by new targets which are the recipients for the subsequent pieces of information – maybe none because no new links are created when t contains no variable. If the term t is a variable y then the link $(x^?, x^!)$ is replaced by the link $(y^?, y^!)$ with the same target and whose source, the (unique) occurrence $x^?$ of variable x , is replaced by the (unique) occurrence $y^?$ of variable y . Therefore, the direction of the flow of information is in both cases preserved: Channels can be viewed as “generalized streams” –that can fork or vanish– through which information is pushed incrementally.

Example 3.4.2. Figure 3.10 shows a guarded attribute grammar that represents two coroutines communicating through lazy streams. Each process alternatively sends and receives data. More precisely the second process sends an acknowledgment –message $?b$ – upon reception of a message sent by the left process. Initially or after reception of an acknowledgment of its previous message the left process can either send a new message or terminate the communication.

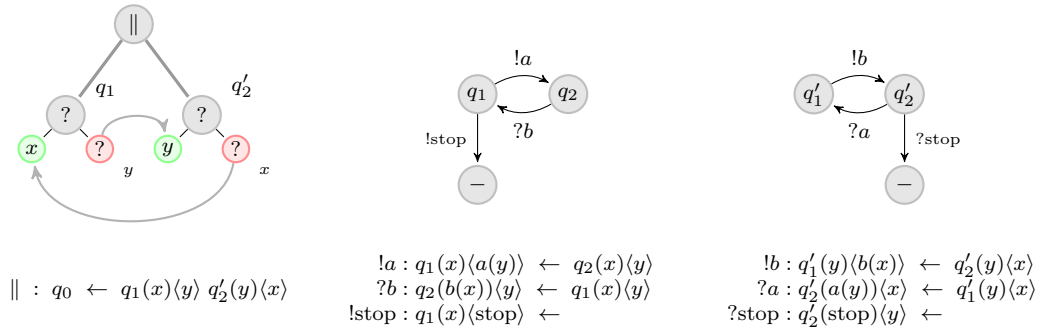
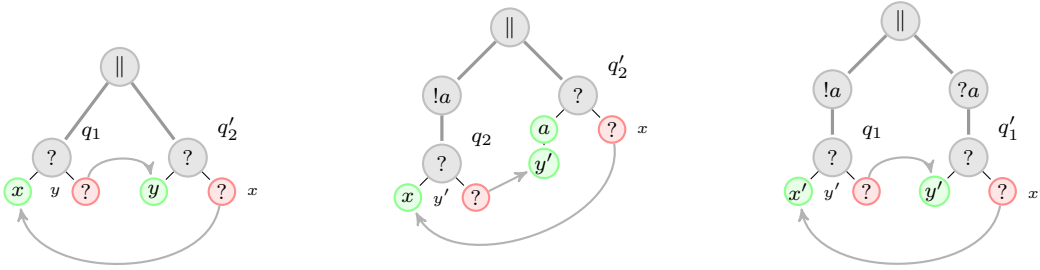


Figure 3.10 – Coroutines with lazy streams

Production $!a : q_1(x')\langle a(y') \rangle \leftarrow q_2(x')\langle y' \rangle$ applies at node X_1 of configuration

$$\Gamma_1 = \{X = X_1 || X_2, \quad X_1 = q_1(x)\langle y \rangle, \quad X_2 = q_2'(y)\langle x \rangle\}$$

shown in Figure 3.11 because its left-hand side $q_1(x')\langle a(y') \rangle$ matches with the definition $q_1(x)\langle y \rangle$

Figure 3.11 – $\Gamma_1[!a/X_1]\Gamma_2[?a/X_2]\Gamma_3$

of X_1 with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = a(y')\}$. We get configuration

$$\Gamma_2 = \left\{ \begin{array}{l} X = X_1 || X_2, \quad X_1 = !a(X_{11}), \\ X_2 = q_2'(a(y'))\langle x \rangle, \quad X_{11} = q_2(x)\langle y' \rangle \end{array} \right\}$$

shown on the middle of Figure 3.11.

Production $?a : q_2'(a(y))\langle x' \rangle \leftarrow q_1'(y)\langle x' \rangle$ applies at node X_2 of Γ_2 because its left-hand side $q_2'(a(y))\langle x' \rangle$ matches with the definition $q_2'(a(y'))\langle x \rangle$ of X_2 with $\sigma_{in} = \{y = y'\}$ and $\sigma_{out} = \{x = x'\}$. We get configuration

$$\Gamma_3 = \left\{ \begin{array}{l} X = X_1 || X_2, \quad X_1 = !a(X_{11}), \quad X_2 = ?a(X_{21}), \\ X_{11} = q_2(x')\langle y' \rangle, \quad X_{21} = q_1'(y')\langle x' \rangle \end{array} \right\}$$

shown on the right of Figure 3.11.

The corresponding acknowledgment may be sent and received leading to configuration

$$\Gamma_5 = \Gamma \cup \{X_{111} = q_1(x)\langle y \rangle, X_{211} = q'_2(y)\langle x \rangle\}.$$

$$\text{where } \Gamma = \left\{ \begin{array}{l} X = X_1 \| X_2, X_1 = !a(X_{11}), X_2 = ?a(X_{21}), \\ X_{21} = !b(X_{211}), X_{11} = ?b(X_{111}) \end{array} \right\}.$$

The process on the left may decide to end communication by applying production !stop : $q_1(x')\langle \text{stop} \rangle \leftarrow$ at X_{111} with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = \text{stop}\}$ leading to configuration

$$\Gamma_6 = \Gamma \cup \{X_{111} = !\text{stop}, X_{211} = q'_2(\text{stop})\langle x \rangle\}.$$

The reception of this message by the process on the right corresponds to applying production ?stop : $q'_2(\text{stop})\langle y \rangle \leftarrow$ at X_{211} with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = y\}$ leading to configuration

$$\Gamma_7 = \Gamma \cup \{X_{111} = !\text{stop}, X_{211} = ?\text{stop}\}.$$

Note that variable x appears in an input position in Γ_6 and has no corresponding output occurrence. This means that the value of x is not used in the configuration. When production ?stop is applied in node X_{211} variable y is substituted to x . Variable y has an output occurrence in production ?stop and no input occurrence meaning that the corresponding output attribute is not defined by the semantic rules. As a consequence, this variable simply disappears in the resulting configuration Γ_7 . If variable x was used in Γ_6 then the output occurrences of x would have been replaced by (output occurrences) of variable y that will remain undefined –no value will be substituted to y in subsequent transformations– until these occurrences of variables may possibly disappear.

End of Example 3.4.2

Chapter 4

Composition, Distribution, and Soundness of GAGs

In this chapter, we investigate some formal properties of guarded attribute grammars. We first turn our attention to the *composition* of autonomous guarded attribute grammars and show that the behaviour of the composed GAG can be recovered from the individual behaviours of its components. We then investigate *distribution*, a classical property of collaborative systems. We consider *input-enabled* GAGs to guarantee that the application of a rule at an open node is a monotonous and confluent operation. This property is instrumental for the distribution of a GAG specification on an asynchronous architecture.

4.1 Composition of Guarded Attribute Grammars

In this section we define the behaviour of potentially non-autonomous guarded attributed grammars to account for systems that call for external services: A guarded attribute grammar for a service may contain terminal symbols, namely symbols that do not occur in the left-hand sides of any of its rules. These terminal symbols are interpreted as calls to external services that are associated with some other guarded attribute grammar. We introduce a composition of guarded attribute grammars and show that the behaviour of the composite guarded attribute grammar can be recovered from the behaviour of its components. We consider the different grammars to be located in separate sites, hence the use of the terms local and distant when referring to variables and/or nodes with respect to a guarded attribute grammar G .

We assume that guarded attribute grammar G has a namespace $ns(G)$ used for the following: the nodes X of its configuration, the variables x occurring in the values of attributes of these nodes, and for references to variables belonging to the active workspaces of other users –its

subscriptions. Hence, we have a name generator that produces unique identifiers for each newly created variable of a configuration. Furthermore, we assume that the name of a variable determines its *location*, namely the active workspace it belongs to. A configuration is given by a set of equations as stated in Definition 3.6 with the following changes:

1. A node associated with a terminal symbol is associated with no equation –it corresponds to a service call that initiates an artifact in a GAG configuration of a distant workspace.
2. Equations of the form $y = x$ state that distant variable y subscribes to the value of local variable x .
3. Equations of the form $Y = X$ in a local configuration state that Y is the distant node that created the artifact rooted at local node X .

Furthermore, we add an input $in(\Gamma)$ and an output $out(\Gamma)$ buffer to each configuration Γ . They contain messages respectively received from and sent to distant locations. A message in the input buffer is one of the following types.

1. $Y = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ tells that distant node Y calls service $s \in \mathbf{axioms}(G)$. When reading this message, we create a new root node X —the root of the artifact associated with the service call. And values t_1, \dots, t_n are assigned to the inherited attributes of node X while the distant variables y_1, \dots, y_m subscribe to the values of its synthesized attributes. We replace variable Y by a dummy variable (wild-card: $_$) when this service call is not invoked from a distant guarded attribute grammar but from an external user of the system.
2. $x = t$ tells that local variable x receives the value t from a subscription created at a distant location.
3. $y = x$ states that distant variable y subscribes to the value of local variable x .

Symmetrically, a message in the output buffer $out(\Gamma)$ is one of the following types.

1. $X = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ tells that local node X calls the external service s –a terminal symbol– with values t_1, \dots, t_n assigned to the inherited attributes. And the local variables y_1, \dots, y_m subscribe to the values of the synthesized attributes of the distant node where the artifact generated by this service call will be rooted.
2. $y = t$ tells that value t is sent to distant variable y according to a subscription made for this variable.
3. $x = y$ states that local variable x subscribes to the value of distant variable y .

The behaviour of a guarded attribute grammar is given by relation $\Gamma \xrightarrow[M]{e} \Gamma'$ stating that event e transforms configuration Γ into Γ' and adds the set of messages M to the output buffer. An event is the application of a rule R to a node X of configuration Γ or the consumption of

a message from its input buffer. Let us start with the former kind of event: $e = R/X$. Let $X = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle \in \Gamma$ and $R = F \rightarrow F_1 \cdots F_k$ be a rule whose left-hand side matches with X producing substitution $\sigma = \mathbf{match}(F, X)$. All variables occurring in the definition of node X are local. We recall that in order to avoid name clashes we rename all the variables of rule R with fresh names. We use the local name generator for that purpose. Hence all variables of R are also local variables – freshly created ones. Therefore, all variables used and defined by substitution σ are local variables. Then we let $\Gamma \xrightarrow[M]{R/X} \Gamma'$ where

$$\begin{aligned} \Gamma' &= \{X = R(X_1, \dots, X_k)\} \\ &\cup \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in N\} \\ &\cup \{X' = F\sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \\ &\cup \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ is a variable}\} \\ &\cup \{Y = X \mid (Y = X) \in \Gamma\} \\ M &= \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in T\} \\ &\cup \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ not a variable}\} \end{aligned}$$

where X_1, \dots, X_k are new names in $ns(G)$. Note that when a distant variable y subscribes to some synthesized attribute of node X , namely $(y = y_i) \in \Gamma$, two situations can occur depending on whether $y_j\sigma$ is a variable or not. When $y_j\sigma = x$ is a (local) variable the subscription $y = y_i$ is replaced by subscription $y = x$: Variable y_i delegates the production of the required value to x . This operation is totally transparent to the location that initiated the subscription. But as soon as some value is produced – $y_j\sigma$ is not a variable – it is immediately sent to the subscribing variable even when this value contains variables: Values are produced and sent incrementally.

Let us now consider the event associated with the consumption of a message $m \in out(\Gamma)$ in the input buffer.

1. If $m = (Y = s(t_1, \dots, t_n)\langle y_1, \dots, y_q \rangle)$ – a call to an external service – then

$$\begin{aligned} \Gamma' &= \Gamma \cup \{\bar{Y} = s(\bar{t}_1, \dots, \bar{t}_n)\langle \bar{y}_1, \dots, \bar{y}_q \rangle\} \\ &\cup \{y_j = \bar{y}_j \mid 1 \leq j \leq q\} \cup \{Y = \bar{Y}\} \end{aligned}$$

where \bar{Y} , the variables \bar{x} for $x \in Var(t_i)$ and the variables \bar{y}_j are new names in $ns(G)$, $\bar{t}_i = t[\bar{x}/x]$, and $M = \{\bar{x} = x \mid x \in Var(t_i) \ 1 \leq i \leq n\}$.

2. If $m = (x = t)$ then $\Gamma' = \Gamma[x = t[\bar{y}/y]]$ where \bar{y} are new names in $ns(G)$ associated with the variables y in t and $M = \{\bar{y} = y \mid y \in Var(t)\}$.
3. If $m = (y = x)$ then $\Gamma' = \Gamma \cup \{y = x\}$ and $M = \emptyset$.

We now define the guarded attribute grammar resulting from the composition of a set of smaller guarded attribute grammars.

Definition 4.1 (Composition of GAG).

Let G_1, \dots, G_p be guarded attribute grammars with disjoint sets of non-terminal symbols such

that each terminal symbol of a grammar G_i that belongs to another grammar G_j must be an axiom of the latter: $T_i \cap S_j = T_i \cap \mathbf{axioms}(G_j)$ where $s \in T_i \cap \mathbf{axioms}(G_j)$ means that grammar G_i uses service s of G_j . Their **composition**, denoted as $G = G_1 \oplus \dots \oplus G_p$, is the guarded attribute grammar whose set of rules is the union of the rules of the G_i s and with set of axioms $\mathbf{axioms}(G) = \cup_{1 \leq i \leq p} \mathbf{axioms}(G_i)$. We say that the G_i are the **local grammars** and G the **global grammar** of the composition. If some axiom of the resulting global grammar calls itself recursively we apply the transformation described in Rem. 3.2.2.

One may also combine this composition with a restriction operator, $G \upharpoonright_{\mathbf{ax}}$, if the global grammar offers only a subset $\mathbf{ax} \subseteq \cup_{1 \leq i \leq p} \mathbf{axioms}(G_i)$ of the services provided by the local grammars.

Note that the set of terminal symbols of the composition is given by

$$T = (\cup_{1 \leq i \leq p} T_i) \setminus (\cup_{1 \leq i \leq p} \mathbf{axioms}(G_i))$$

that is, its set of external services are all external services of a local grammar but those which are provided by some other local grammar in the composition. Note also that the set of non-terminal symbols of the global grammar is the (disjoint) union of the set of non-terminal symbols of the local grammars: $N = \cup_{1 \leq i \leq p} N_i$. This partition can be used to retrieve the local grammar by taking the rules of the global grammar whose sorts on the left-hand side belong to the given equivalence class. Of course, not every partition of the set of non-terminal symbols of a guarded attribute grammar corresponds to a decomposition into local grammars. To decompose a guarded attribute grammar into several components one can proceed as follows:

1. Select a partition $\mathbf{axiom}(G) \subseteq \cup_{1 \leq i \leq n} \mathbf{axioms}_i$ of the set of axioms. These sets are intended to represent the services associated with each of the local grammars.
2. Construct the local grammar G_i associated with services \mathbf{axioms}_i by first taking the rules whose left-hand sides are forms of sort $s \in \mathbf{axioms}_i$, and then iteratively adding to G_i the rules whose left-hand sides are forms of sort $s \in N \setminus \cup_{j \neq i} \mathbf{axioms}_j$ such that s appears in the right-hand side of a rule previously added to G_i .
3. If appropriate, namely when a same rule is copied in several components, rename the non-terminal symbols of the local grammars to ensure that they have disjoint sets of non-terminal symbols.

The above transformation can duplicate rules in G in the resulting composition $\overline{G} = G_1 \oplus \dots \oplus G_n$ but does not radically change the original specification.

Configurations of guarded attribute grammars are enriched with subscriptions –equations of the form $y = x$ – to enable communication between the various sites. One might dispense with equations of the form $Y = X$ in the operational semantics of guarded attribute grammars. But they facilitate the proof of correctness of this composition (Prop. 4.1.1) by easing the

reconstruction of the global configuration from its set of local configurations. Indeed, the global configuration can be recovered as $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ where operator \oplus consists in taking the union of the systems of equations given as arguments and simplifying the resulting system by elimination of the copy rules: We drop each equation of the form $Y = X$ (respectively $y = x$) and replace each occurrence of Y by X (resp. of y by x).

Let $G = G_1 \oplus \cdots \oplus G_p$ be a composition, Γ_i be a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ the corresponding global configuration. Since the location of a variable derives from its identifier we know the location of destination of every message in the output buffer of a component. If M is a set of messages we let $M_i \subseteq M$ denote the set of messages, in M to be forwarded to G_i . Their union $M_i = \cup_{1 \leq i \leq p} M_i$ is the set of *internal* messages that circulate between the local grammars. The remainder $M_G = M \setminus M_i$ is the set of messages that remain in the output buffer of the global grammar –the *global* messages.

The joint dynamics of the local grammars can be derived as follows from their individual behaviours, where e stands for R/X or a message m :

1. If $\Gamma_i \xrightarrow[M]{e} \Gamma'_i$ then $\Gamma \xrightarrow[M]{e} \Gamma'$ with $\Gamma_j = \Gamma'_j$ for $j \neq i$.
2. If $\Gamma \xrightarrow[M]{e} \Gamma'$ and $\Gamma' \xrightarrow[M']{m} \Gamma''$ for $m \in M$ then $\Gamma \xrightarrow[M \setminus \{m\} \cup M']{e} \Gamma''$.

The correctness of the composition is given by the following proposition. It states that (i) every application of a rule can immediately be followed by the consumption of the local messages generated by it, and (ii) the behaviour of the global grammar can be recovered from the joint behaviour of its components where all internal messages are consumed immediately.

Proposition 4.1.1. *Let $G = G_1 \oplus \cdots \oplus G_p$ be a composition, Γ_i a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \cdots \oplus \Gamma_p$ the corresponding global configuration. Then*

1. $\Gamma \xrightarrow[M]{R/X} \Gamma' \implies \exists! \Gamma''$ such that $\Gamma \xrightarrow[M_G]{R/X} \Gamma''$
2. $\Gamma \xrightarrow[M_G]{e} \Gamma' \iff \Gamma \xrightarrow[M_G]{e} \Gamma'$

Proof. The first statement follows from the fact that relation $\Gamma \xrightarrow[M]{e} \Gamma'$ is deterministic (M and Γ' are uniquely defined from Γ and e) and the application of a rule produces, directly or indirectly, a finite number of messages.

1. If m is of the form $Y = s(t_1, \dots, t_n) \langle y_1, \dots, y_n \rangle$, then consuming m results in adding new equations to the local configuration that receives it, and generating a set of messages of the form $\bar{x} = x$ that can hence be consumed by the location that will receive them without generating new messages (case 3 below).
2. If $m = (x = t)$, then consumption of the message results in production of new variables, and a new (finite) set of messages of the form $\bar{y} = y$ that can then be consumed by the location that sent m without producing any new message.

3. If $m = (y = x)$ then consuming the message results in adding an equation $y = x$ to the local configuration without generating any new message.

The second statement follows from the fact that the construction of a global configuration $\Gamma = \Gamma_1 \oplus \dots \oplus \Gamma_p$ amounts to consuming all the local messages between the components.

Q.E.D.

Corollary 4.1.2. *Let $G = G_1 \oplus \dots \oplus G_p$ be a composition where G is an autonomous guarded attribute grammar and Γ be a configuration of G . Then*

$$\Gamma[R/X]\Gamma' \iff \Gamma \xrightarrow[\emptyset]{R/X} \Gamma'$$

A configuration is *stable* when all internal messages are consumed. The behaviour of the global grammar is thus given as the restriction of the joint behaviour of the components to the set of stable configurations. This amounts to imposing that every event of the global configuration is immediately followed by the consumption of the internal messages generated by the event. However, if the various sites are distributed at distant locations on an asynchronous architecture, one can never guarantee that no internal message remains in transit, or that some message is received but not yet consumed in some distant location. In order to ensure a correct distribution, we therefore need a monotony property stating that (i) a locally enabled rule cannot become disabled by the arrival of a message and (ii) local actions and incoming messages can be swapped. We identify in Section 4.2 a class of guarded attribute grammars having this monotony property and which thus guarantees a safe implementation of distributed active workspaces.

4.2 Distribution of a Guarded Attribute Grammar

We say that rule R is **triggered** in node X if substitution σ_{in} –given in def. 3.9– is defined: Patterns p_i match the data d_i . As shown by the following example one can usually suspect a flaw in a specification when a triggered transition is not enabled due to the fact that the system of equations $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ is cyclic.

Example 4.2.1. Let us consider the guarded attribute grammar given by the following rules:

$$\begin{aligned} P : & \quad s_0(\langle \rangle) \rightarrow s_1(a(x))\langle x \rangle \quad s_2(x)\langle \rangle \\ Q : & \quad s_1(y)\langle a(y) \rangle \rightarrow \\ R : & \quad s_2(a(z))\langle \rangle \rightarrow \end{aligned}$$

Applying P in node X_0 of configuration $\Gamma_0 = \{X_0 = s_0()\langle\rangle\}$ leads to configuration

$$\Gamma_1 = \{X_0 = P(X_1, X_2); X_1 = s_1(a(x))\langle x \rangle; X_2 = s_2(x)\langle\rangle\}$$

Rule Q is triggered in node X_1 with $\sigma_{in} = \{y = a(x)\}$ but the occur check fails because variable x occurs in $a(y)\sigma_{in} = a(a(x))$. Alternatively, we could drop the occur check and instead adapt the fixed-point semantics for attribute evaluation defined in [20, 73] in order to cope with infinite data structures. More precisely, we let σ_{out} be defined as the least solution of system of equations $\{y_i = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ —assuming these equations are guarded, i.e., that there is no cycle of copy rules in the link graph of any accessible configuration. In that case the infinite tree a^ω is substituted to variable x and the unique maximal computation associated with the grammar is given by the infinite tree $P(Q, R^\omega)$. In Definition 3.10 we have chosen to restrict to finite data structures which seems a reasonable assumption in view of the nature of systems we want to model. The occur check is used to avoid recursive definitions of attribute values. The given example, whose most natural interpretation is given by fixed point computation, should in that respect be considered as ill-formed. And indeed, this guarded attribute grammar is not *sound* —a notion presented in Section 4.2— because configuration Γ_1 is not closed (it still contains open nodes) but yet it is a terminal configuration that enables no rule. We say that this configuration is not *deadlock free* since it represents a case that cannot be terminated.

End of Example 4.2.1

The fact that triggered rules are not enabled can also impact the distributability of a grammar as shown by the following example.

Example 4.2.2. Let us consider the GAG with the following rules:

$$\begin{aligned} P &: s()\langle\rangle && \rightarrow s_1(x)\langle y \rangle \ s_2(y)\langle x \rangle \\ Q &: s_1(z)\langle a(z) \rangle && \rightarrow \\ R &: s_2(u)\langle a(u) \rangle && \rightarrow \end{aligned}$$

Rule P is enabled in configuration $\Gamma_0 = \{X_0 = s()\langle\rangle\}$ with $\Gamma_0[P/X_0]\Gamma_1$ where

$$\Gamma_1 = \{X_0 = P(X_1, X_2); X_1 = s_1(x)\langle y \rangle, X_2 = s_2(y)\langle x \rangle\}.$$

In configuration Γ_1 rules Q and R are enabled in nodes X_1 and X_2 respectively with $\Gamma_1[Q/X_1]\Gamma_2$ where

$$\Gamma_2 = \{X_0 = P(X_1, X_2); X_1 = Q, X_2 = s_2(a(x))\langle x \rangle\}$$

and $\Gamma_1[R/X_2]\Gamma_3$ where

$$\Gamma_3 = \{X_0 = P(X_1, X_2); X_1 = s_2(a(y))\langle y \rangle, X_2 = R\}$$

Now rule R is triggered but not enabled in node X_2 of configuration Γ_2 because of the cyclicity of $\{x = a(a(x))\}$. Similarly, rule Q is triggered but not enabled in node X_3 of configuration Γ_3 . There is a conflict between the application of rules R and Q in configuration Γ_1 . When the grammar is distributed in such a way that X_1 and X_2 have distinct locations, the specification is not implementable. *End of Example 4.2.2*

We first tackle the problem of safe distribution of a GAG specification on an asynchronous architecture by limiting ourselves to standalone systems. Hence to autonomous guarded attribute grammars. At the end of the section we show that this property can be verified in a modular fashion if the grammar is given as the composition of local (and thus non-autonomous) grammars.

Definition 4.2 (Accessible Configurations).

Let G be an autonomous guarded attribute grammar. A **case** $c = s(t_1, \dots, t_n)\langle x_1, \dots, x_m \rangle$ is a ground instantiation of service s , an axiom of the grammar, i.e., the values t_i of the inherited attributes are ground terms. It means that it is a service call which already contains all the information coming from the environment of the guarded attribute grammar. An **initial configuration** is any configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with case c . An **accessible configuration** is any configuration accessible from an initial configuration.

Substitution σ_{in} , given by pattern matching, is monotonous w.r.t. incoming information and thus it causes no problem for a distributed implementation of our model. However, substitution σ_{out} is not monotonous since it may become undefined when information coming from a distant location makes the match of output attributes a cyclic set of equations, as illustrated by example 4.2.2.

Definition 4.3. An autonomous guarded attribute grammar is **input-enabled** if every rule that is triggered in an accessible configuration is also enabled.

If every form $s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ occurring in some reachable configuration is such that variables y_j do not appear in expressions d_i then by Remark 3.3.2 the guarded attribute grammar is input-enabled —moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ for every enabled rule. This property is clearly satisfied for guarded L-attributed grammars which consequently constitute a class of input-enabled guarded attribute grammars.

Definition 4.4 (L-attributed Grammars). A guarded attribute grammar is left-attributed, in short a LGAG, if any variable that is used in an inherited position in some form F of the right-hand side of a rule is either a variable defined in a pattern in the left-hand side of the rule or a variable occurring at a synthesized position in a form which appears at the left of F , i.e., inherited information flows from top-to-bottom and left-to-right between sibling nodes.

We call the *substitution induced by a sequence* $\Gamma[*]\Gamma'$ the corresponding composition of the various substitutions associated respectively with each of the individual steps in the sequence. If X is an open node in both Γ and Γ' , i.e., no rules are applied at node X in the sequence, then we get $X = s(d_1\sigma, \dots, d_n\sigma)\langle y_1, \dots, y_m \rangle \in \Gamma'$ where

$$X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle \in \Gamma$$

and σ is the substitution induced by the sequence.

Proposition 4.2.3 (Monotony).

Let Γ be an accessible configuration of an input-enabled GAG, $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle \in \Gamma$ and σ be the substitution induced by some sequence starting from Γ . Then

$$\Gamma[P/X]\Gamma' \quad \text{implies} \quad \Gamma\sigma[P/X]\Gamma'\sigma.$$

Proof. Direct consequence of Definition 3.3 due to the fact that

1. $\mathbf{match}(p, d\sigma) = \mathbf{match}(p, d)\sigma$, and
2. $\mathbf{mgu}(\{y_j = u_j\sigma \mid 1 \leq j \leq m\}) = \mathbf{mgu}(\{y_j = u_j \mid 1 \leq j \leq m\})\sigma$.

The former is trivial, and the latter follows by induction on the length of the computation of the most general unifier –relation \Rightarrow^* using rule (5) only of Recall 3.3.4. Note that the assumption that the guarded attribute grammar is input-enabled is crucial because in the general case it could happen that the set $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ satisfies the occur check whereas the set $\{y_j = u_j(\sigma_{in}\sigma) \mid 1 \leq j \leq m\}$ does not satisfy the occur check. *Q.E.D.*

Proposition 4.2.3 is instrumental for the distributed implementation of guarded attribute grammars. Namely it states that new information coming from a distant asynchronous location refining the value of some input occurrences of variables of an enabled rule do not prevent the rule to apply. Thus, a rule that is locally enabled can freely be applied regardless of information that might further refine the current partial configuration. It means that conflicts arise only from the existence of two distinct rules enabled in the same open node. Hence the only form of non-determinism corresponds to the decision of a stakeholder to apply one particular rule among those enabled in a configuration. This is expressed by the following confluence property.

Corollary 4.2.4. Let Γ be an accessible configuration of an input enabled GAG. If $\Gamma[P/X]\Gamma_1$ and $\Gamma[Q/Y]\Gamma_2$ with $X \neq Y$ then $\Gamma_2[P/X]\Gamma_3$ and $\Gamma_1[Q/Y]\Gamma_3$ for some configuration Γ_3 .

Note that, by Corollary 4.2.4, the artifact contains a full history of the case in the sense that one can reconstruct from the artifact the complete sequence of applications of rules leading to the resolution of the case —up to the commutation of independent elements in the sequence.

Remark 4.2.5. We might have considered a more symmetrical presentation in Definition 3.3 by allowing patterns for synthesized attributes in the right-hand sides of rules with the effect of creating forms in a configuration with patterns in their co-arguments. These patterns would express constraints on synthesized values. This extension could be acceptable if one sticks to purely centralized models. However, as soon as one wants to distribute the model on an asynchronous architecture, one cannot avoid such a constraint to be further refined due to a transformation occurring in a distant location. Then the monotony property (Proposition 4.2.3) is lost: A locally enabled rule can later be disabled when a constraint on a synthesized value gets a refined value. This is why we required synthesized attributes in the right-hand side of a rule to be given by plain variables in order to prohibit constraints on synthesized values.

End of Remark 4.2.5

It is difficult to verify input-enabledness as the whole set of accessible configurations is involved in this condition. Nevertheless, one can find a sufficient condition for input enabledness, similar to the strong non-circularity of attribute grammars [24], that can be checked by a simple fix-point computation.

Definition 4.5. Let s be a sort of a guarded attribute grammar with n inherited attributes and m synthesized attributes. We let $(j, i) \in SI(s)$ where $1 \leq i \leq n$ and $1 \leq j \leq m$ if there exists $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle \in \Gamma$ where Γ is an accessible configuration and $y_j \in d_i$. If R is a rule with left-hand side $s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ we let $(i, j) \in IS(R)$ if there exists a variable $x \in \text{Var}(R)$ such that $x \in \text{Var}(d_i) \cap \text{Var}(u_j)$. The guarded attribute grammar G is said to be **acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $G(s, R) = SI(s) \cup IS(R)$ is acyclic.

Proposition 4.2.6. An acyclic guarded attribute grammar is input-enabled.

Proof. Suppose R is triggered in node X with substitution σ_{in} such that $y_j \in u_i\sigma_{in}$ then $(i, j) \in G(s, R)$. Then the fact that occur check fails for the set $\{y_j \mid 1 \leq j \leq m\}$ entails that one can find a cycle in $G(s, R)$. *Q.E.D.*

Relation $SI(s)$ still considers the whole set of accessible configurations. The following definition provides an over-approximation of this relation given by a fixpoint computation.

Definition 4.6. The **graph of local dependencies** of a rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ is the directed graph $GLD(R)$ that records the data dependencies between the occurrences of attributes given by the semantic rules. We designate the occurrences of attributes of R as follows: We let $k(i)$ (respectively $k(j)$) denote the occurrence of the i^{th} inherited attribute (resp. the j^{th} synthesized attribute) in F_k . If s is a sort with n inherited attributes and m synthesized attributes, we

define the relations $\overline{IS}(s)$ and $\overline{SI}(s)$ over $[1, n] \times [1, m]$ and $[1, m] \times [1, n]$ respectively as the least relations such that:

1. For every rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ where form F_i is of sort s_i and for every $k \in [1, \ell]$

$$\{(j, i) \mid (k\langle j \rangle, k\langle i \rangle) \in GLD(R)^k\} \subseteq \overline{SI}(s_k)$$

where graph $GLD(R)^k$ is given as the transitive closure of

$$GLD(R) \cup \left\{ (0\langle j \rangle, 0\langle i \rangle) \mid (j, i) \in \overline{SI}(s_0) \right\} \\ \cup \left\{ (k'\langle i \rangle, k'\langle j \rangle) \mid k' \in [1, \ell], k' \neq k, (i, j) \in \overline{IS}(s_{k'}) \right\}$$

2. For every rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ where form F_i is of sort s_i

$$\{(i, j) \mid (0\langle i \rangle, 0\langle j \rangle) \in GLD(R)^0\} \subseteq \overline{IS}(s_0)$$

where graph $GLD(R)^0$ is given as the transitive closure of

$$GLD(R) \cup \left\{ (k\langle i \rangle, k\langle j \rangle) \mid k \in [1, \ell], (i, j) \in \overline{IS}(s_k) \right\}$$

The guarded attribute grammar G is said to be **strongly-acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $\overline{G}(s, R) = \overline{SI}(s) \cup \overline{IS}(R)$ is acyclic.

Proposition 4.2.7. *A strongly-acyclic guarded attribute grammar is acyclic and hence input-enabled.*

Proof. The proof is analogous to the proof that a strongly non-circular attribute grammar is non-circular, and it goes as follows. We let $(i, j) \in \overline{IS}(s)$ when $\text{Var}(d_i\sigma) \cap \text{Var}(y_j\sigma) \neq \emptyset$ for some form $F = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of sort s and where σ is the substitution induced by a firing sequence starting from configuration $\{X = F\}$. Then we show by induction on the length of the firing sequence leading to the accessible configuration that $\overline{IS}(s) \subseteq \overline{IS}(s)$ and $\overline{SI}(s) \subseteq \overline{SI}(s)$. Q.E.D.

Note that the following two inclusions are strict

$$\text{strongly-acyclic GAG} \subsetneq \text{acyclic GAG} \subsetneq \text{input enabled GAG}$$

Indeed the reader may easily check that the guarded attribute grammar with rules

$$\left\{ \begin{array}{l} A(x)\langle z \rangle \rightarrow B(a(x, y))\langle y, z \rangle \\ B(a(x, y))\langle x, y \rangle \rightarrow \end{array} \right.$$

is cyclic and input-enabled whereas guarded attribute grammar with rules

$$\begin{cases} A(x)\langle z \rangle \rightarrow B(y, x)\langle z, y \rangle \\ A(x)\langle z \rangle \rightarrow B(x, y)\langle y, z \rangle \\ B(x, y)\langle x, y \rangle \rightarrow \end{cases}$$

is acyclic but not strongly-acyclic. Attribute grammars arising from real situations are almost always strongly non-circular so that this assumption is not really restrictive. Similarly, we are confident that most of the guarded attribute grammars that we shall use in practise will be input-enabled and that most of the input-enabled guarded attribute grammars are in fact strongly-acyclic. Thus, most of the specifications are distributable and in most cases, this can be proven by checking strong non-circularity.

Let us conclude this section by addressing the modularity of strong-acyclicity. This property (see Def. 4.6) however was defined for autonomous guarded attribute grammars viewed as standalone applications. Here, the initial configuration is associated with a case $c = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ introduced by the external environment. And the information transmitted to the inherited attributes are given by ground terms. Even though one can imagine that these values are introduced gradually they do not depend on the values that will be returned to the subscribing variables y_1, \dots, y_m . It is indeed reasonable to assume that when a user enters a new case in the system he/she instantiates the inherited information and then waits for the returned values. Things go differently if the autonomous guarded attribute grammar is not a standalone application but a component in a larger specification. In that case, a call to the service provided by this component is of the form $s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ where the values transmitted to the inherited attributes may depend (directly or indirectly) on the subscribing variables. Definition 4.6 should be amended to incorporate these dependencies and strong-acyclicity can be lost. Therefore, a component which is strongly-acyclic when used as a standalone application might lose this property when it appears as an individual component of a larger specification. Thus, if the component is already implemented as a collection of sub-components distributed on an asynchronous architecture, the correctness of this distribution can be lost if the component takes part in a larger system.

To avoid this pitfall, we follow a standard contract-based approach, where each component can be developed independently as long as it conforms to constraints given by assume/guarantee conditions. Assuming some properties of the environment, this approach allows to preserve properties of assembled components. In our case, we show that strong-acyclicity is preserved by composition.

Definition 4.7. *Let s be a sort with n inherited attributes and m synthesized attributes. We let $\mathbf{IS}(s) = [1, n] \times [1, m]$ and $\mathbf{SI}(s) = [1, m] \times [1, n]$ denote the set of (potential) dependencies between inherited and synthesized attributes of s . Let G be a guarded attribute grammar with*

axioms s_1, \dots, s_k and terminal symbols $s'_1, \dots, s'_{k'}$. An **assume/guarantee condition** for G is a pair $(a, g) \in \mathbf{AG}(G)$ with $a \in \mathbf{SI}(s_1) \times \dots \times \mathbf{SI}(s_k) \times \mathbf{IS}(s'_1) \times \dots \times \mathbf{IS}(s'_{k'})$ and $g \in \mathbf{IS}(s_1) \times \dots \times \mathbf{IS}(s_k) \times \mathbf{SI}(s'_1) \times \dots \times \mathbf{SI}(s'_{k'})$. Equivalently it is given by the data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ for $s \in \mathbf{axioms}(G) \cup T$. The guarded attribute grammar G is strongly-acyclic w.r.t. assume/guarantee condition (a, g) if the modified fixed-point computation of Def. 4.6, where constraints $SI(s) \subseteq \overline{SI(s)}$ for $s \in \mathbf{axioms}(G)$ and $IS(s) \subseteq \overline{IS(s)}$ for $s \in T$ are added, allows to conclude strong-acyclicity with $\overline{IS(s)} \subseteq IS(s)$ for $s \in \mathbf{axioms}(G)$ and $\overline{SI(s)} \subseteq SI(s)$ for $s \in T$.

The data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ give an over-approximation of the attribute dependencies, the so-called 'potential' dependencies, for the axioms and the terminal symbols. They define a *contract* of the guarded attribute grammar. This contract splits into *assumptions* about its environment –SI dependencies for the axioms and IS dependencies for the terminal symbols– and *guarantees* offered in return to the environment –IS dependencies for the axioms and SI dependencies for the terminal symbols. Thus strongly-acyclicity of a guarded attribute grammar G w.r.t. assume/guarantee condition (a, g) means that when the environment satisfies the assume condition, grammar G is strongly-acyclic and satisfies the guarantee condition.

The following result states the modularity of strong-acyclicity.

Proposition 4.2.8. *Let $G = G_0 \oplus \dots \oplus G_p$ be a composition of guarded attribute grammars. Let $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ be assumptions on the (potential) dependencies between attributes where sort s ranges over the set of axioms and terminal symbols of the components G_i –thus containing also the axioms and terminal symbols of global grammar G . These constraints restrict to assume/guarantee conditions $(a_i, g_i) \in \mathbf{AG}(G_i)$ for every local grammar and for the global grammar as well: $(a, g) \in \mathbf{AG}(G)$. Then G is strongly-acyclic w.r.t. (a, g) when each local grammar G_i is strongly-acyclic w.r.t. (a_i, g_i) .*

Proof. The fact that the fixed-point computation for the global grammar can be computed component-wise follows from the fact that for each local grammar no rule applies locally to a terminal symbol s and consequently rule 3 in Def. 4.6 never applies for s and the value $\overline{SI(s)}$ is left unmodified during the fixpoint computation, it keeps its initial value $SI(s)$. Similarly, rule 2 in Def. 4.6 never applies for an axiom s and $\overline{IS(s)}$ keeps its initial value $IS(s)$. *Q.E.D.*

Conversely if the global grammar is strongly-acyclic w.r.t. some assume/guarantee condition (a, g) then the values of $\overline{SI(s)}$ and $\overline{IS(s)}$ produced at the end of the fixpoint computation allows to complement the assume/guarantee conditions with respect to which the local grammars are strongly-acyclic. The issue is how to guess some correct assume/guarantee conditions in the

first place. One can imagine that some knowledge about the problem at hand can help to derive the potential attribute dependencies, and that we can use them to type the components for their future reuse in larger specifications. In many cases however there is no such dependencies and the assume/guarantee conditions are given by empty relations.

Definition 4.8. A composition $G = G_0 \oplus \dots \oplus G_p$ of guarded attribute grammars is **distributable** if each local grammar (and hence also the global grammar) is strongly-acyclic w.r.t. the empty assume/guarantee condition.

This condition might seem rather restrictive but it is not. Indeed $(i, j) \notin IS(s)$ (similarly for $(i, j) \notin SI(s)$) does not mean that the j^{th} synthesized attribute does not depend on the value received by the i^{th} inherited attribute –the inherited value influences the behaviour of the component and hence has an impact on the values that will be returned in synthesized attributes. It rather says that one should not return in the value of a synthesized attribute some data directly extracted from the value of inherited attributes, which is the most common situation. The emptiness of assume/guarantee gives us a criterion for a distributable decomposition of a guarded attribute grammar: It indicates the places where a specification can safely be split into smaller pieces. We shall denote a distributable composition as:

$$\begin{aligned}
 G &= \langle G_1, \dots, G_p \rangle \\
 \text{where } G_1 &:: \% \text{definition of } G_1 \\
 &\vdots \\
 G_p &:: \% \text{definition of } G_p
 \end{aligned}$$

4.3 Soundness of Guarded Attribute Grammars

In this chapter, we are interested in checking the correctness of GAG specifications in terms of termination and soundness. A specification is *sound* if every case can reach completion no matter how its execution started. Recall from Def. 4.2 that a case $c = s_0(t_1, \dots, t_n)\langle x_1, \dots, x_m \rangle$ is a ground instantiation of service s_0 , an axiom of the grammar. And, an accessible configuration is any configuration accessible from a configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with a case c (an initial configuration).

Definition 4.9. A configuration is **closed** if its artifact contains only closed nodes. An autonomous guarded attribute grammar is **sound** if a closed configuration is accessible from any accessible configuration.

We consider the finite sequences $(\Gamma_i)_{0 < i \leq n}$ and the infinite sequences $(\Gamma_i)_{0 < i < \omega}$ of accessible configurations such that $\Gamma_i \langle \rangle \Gamma_{i+1}$. A finite and maximal sequence is said to be **terminal**.

Hence a terminal sequence leads to a configuration that enables no rule. Soundness can then be rephrased by the two following conditions.

1. Every terminal sequence leads to a closed configuration.
2. Every configuration on an infinite sequence also belongs to some terminal sequence.

An infinite sequence of accessible configurations is the consequence of a recursive underlying GAG, that is, one that admits loops. In the latter, there exist at least one sort that recursively derives to itself ($s \rightarrow^* \dots s \dots$). Transforming an infinite sequence of configurations into a finite equivalence requires that two conditions be met:

1. for every sort s in the underlying GAG such that s can be recursively derived from itself ($s \rightarrow^* \dots s \dots$), there exist at least one other derivation of s which never derives to itself ($s \rightarrow^* \dots s' \dots \not\rightarrow s$).

This condition is not sufficient given that it does not guarantee that the loop-free derivation will eventually be taken. We could ignore this fact and suppose that the user will manually chose to use the loop-free derivation at some point. With this assumption however, we are not able to answer by a strict Yes or No to whether the model terminates.

2. the set of inherited attributes admits a well-founded relation [28, 45], that is, the set is strictly decreasing and has a lower bound. This guarantees that each iteration brings us closer to leaving the loop, and that eventually the loop can be exited (when the lower bound is reached). Defining such a relation on terms is not only non-trivial but it adds a non-negligible layer of complexity to GAG based process modelling.

Lemma 4.3.1. *All sequences of configurations, $\Gamma_i[\]\Gamma_{i+1}$, for a GAG with no loops, are finite.*

Proof. of Lem. 4.3.1

It follows from the structure of guarded attribute grammars and their finite set of business rules. First we transform all non-autonomous GAGs into autonomous GAGs by safely adding a terminal business rule for each of the terminal symbols of the grammar. The absence of loops guarantees that for each node n of sort s of the derivation graph T , s does not appear in the subtree of n .

Eventually the leaves of T contain only empty nodes, originating from terminal rules with empty right hand sides. *Q.E.D.*

Soundness of guarded attribute grammars has however been proven undecidable in [11] using a simple encoding of Minsky machines, on simplified specifications: guards of depth at most one, deterministic systems with only two inherited attributes and no synthesized attributes. Still in [11], (weak) soundness was proven if a restricted form of composition is employed in GAG

specifications – *hierarchical composition*. Hierarchical composition, restricts the instantiation and orchestration of service calls to the so-called *connector*. All possible service orchestrations are encoded in the connector. These restrictions not only complexify the GAG model, but also does not naturally support the dynamic processes paradigm.

In this chapter, we introduce a representation of a *complete artifact* – one in which all possible executions of a case are represented – and use it to define a class of guarded attribute grammars on which soundness can be verified in polynomial time.

4.3.1 Preliminaries

Our demonstration exploits the GAG operational semantics with its two decision points: the automated filtering of rules at open nodes and the manual choice of a rule to apply at a node by the user. We start by defining a method to extract and manipulate the guards of business rules as hypotheses posed on data from the environment. Then we introduce the concept of *complete-artifact* and its building blocks.

4.3.1.1 GAG Hypotheses

Definition 4.10. *Given a task t of sort s and a set of rules of the same sort, a **hypothesis** (h) on the environment of t is a (pattern-based) condition on variables at inherited positions of the rules whose values are provided exclusively by the environment.*

Variables in inherited attributes whose values are provided by sibling tasks are excluded from the hypothesis. A hypothesis therefore is a conjunction of atomic terms (substitutions) of the form $x = t$, which when satisfied enables a set of business rules.

Example 4.3.2. *Extracting Hypotheses from Business Rules*

Let s be an axiom of a GAG with three rules, R1, R2 and R3, all of sort s , whose profiles (left hand sides) are respectively, R1 : $s(a(x))\langle y \rangle \rightarrow$, R2 : $s(a(b(x)))\langle y \rangle \rightarrow$, and R3 : $s(b(x))\langle y \rangle \rightarrow$.

Let att_1 denote the lone inherited attribute in these rules. By simply observing the guards, we distinguish the following three hypotheses on att_1 :

1. h_1 : $att_1 = a(x)$, which enables rule R1, written $(h_1, \{R1\})$
2. h_2 : $att_1 = a(b(x))$, which enables rules R1 and R2, written $(h_2, \{R1, R2\})$
3. h_3 : $att_1 = b(x)$, which enables rule R3. written $(h_3, \{R3\})$

End of Example 4.3.2

4.3.1.2 From Hypotheses to Disjoint Clauses

In this section, we define an approach to compute pairwise disjoint clauses from basic hypotheses. Such clauses are useful to eliminate all sources of ambiguity that may arise during simulation when data from the environment is used to choose the branch of the derivation tree to scan.

Let $H = \{(h_1, R_1), (h_2, R_2), \dots, (h_n, R_n)\}$ be the initial set of hypothesis extracted from business rules. We start by defining an associative and commutative composition operation of two hypotheses $(H_i \odot H_j)$, $H_i \in H$ and $H_j \in H$.

$$H_i \odot H_j = \begin{cases} (mgu(h_i, h_j), R_i \cup R_j) & : \text{if } mgu(h_i, h_j) \text{ succeeds} \\ \emptyset & : \text{Otherwise} \end{cases}$$

Notice the use of the *mgu* (most general unifier) operation defined in Section 3.3.2 (cf. Recall 3.3.4). We denote by $\odot S$ the associative composition of $S \subseteq H$.

We also define an exclusivity operation between a subset of $S_i \subseteq H$ and another subset $S_j \subseteq H$ with $i \neq j$ and $Dom(S_i) \cap Dom(S_j) \neq \emptyset$. Where $Dom(S_i)$ (resp. $Dom(S_j)$) constitutes the set of variables appearing in hypothesis S_i (resp. S_j).

$$S_i \ominus S_j = \bigvee_{x \in Dom(S_i) \cap Dom(S_j)} \{t \stackrel{?}{=} u \mid x = t \in H_i, x = u \in H_j\}$$

This states that, there should exist at least one shared variable $x \in Dom(S_i) \cap Dom(S_j)$ whose value is different in the two clauses. The difference between two values ($t \stackrel{?}{=} u$) is computed by the following rewriting rules:

$$t \stackrel{?}{=} u \mid x \in Dom(S_i), x = t \in H_i, x = u \in H_j = \begin{cases} a(\dots) \stackrel{?}{=} b(\dots) & \rightarrow x \neq b(\dots) \\ a(t_1, \dots, t_m) \stackrel{?}{=} a(u_1, \dots, u_m) & \rightarrow t_i \stackrel{?}{=} u_i \\ a \stackrel{?}{=} a & \rightarrow \emptyset \end{cases}$$

This exclusivity operation computes the set of inequalities that will need to be associated with each clause S_i to render it disjoint from all the other clauses.

Finally, we define the union of two subsets of H , $S_i \oplus S_j$, as follows: the disjoint union of all the hypothesis in all the elements of S_i and S_j coupled with the disjoint union of all the business rules all elements of S_i and S_j .

$$S_i \oplus S_j = \left(\left(\bigcup_{h_k \in S_i} h_k \uplus \bigcup_{h_l \in S_j} h_l \right), \left(\bigcup_{R_k \in S_i} R_k \uplus \bigcup_{R_l \in S_j} R_l \right) \right)$$

Algorithm 1 uses the composition, exclusivity, and union operations defined above to compute the set of pairwise disjoint clauses δ from the initial set of hypotheses H . Each δ_i is of the form $\delta_i = \bigwedge(\dots, x = t, \dots) \wedge (\bigvee(\dots, x \neq u_1, \dots) \wedge \bigvee(\dots, x \neq u_2, \dots) \wedge \dots)$, and it can easily be proven that using the identities of Propositional Calculus, like the double negation elimination, the De Morgan laws and distributivity, one can transform any clause into an equivalent formula in Disjunctive Normal Form (DNF). Also, the operation on Line 6 ensures that the final set of clauses contains only maximal subsets of H by replacing any two equivalent (containing exactly the same values for shared variables) subsets by their union.

Algorithm 1: Computing Pair-wise Disjoint Clauses from Hypotheses

Data: Set of initial Hypotheses $H = \{(h_1, R_1), (h_2, R_2), \dots, (h_n, R_n)\}$

Result: Set of disjoint clauses $\delta = \{\delta_1, \dots, \delta_k\}$

```

1 Compute  $S = \{S_1, S_2, \dots, S_k\}$  with  $S_i \subseteq H$  and  $\odot S \neq \emptyset$ ;
2 for  $S_i \in S$  do
  // compute  $\psi = \{\psi_1, \psi_2, \dots, \psi_k\}$ 
3    $\psi_i = \top$ ;
4   for  $S_j \in \overline{S_i}$  do
5     if  $S_i \ominus S_j = \emptyset$  then
6       |  $S = (S \setminus \{S_i, S_j\}) \cup (S_i \oplus S_j)$  // Ensures that each  $S_i$  is maximal
7     else
8       |  $\psi_i = \psi_i \wedge (S_i \ominus S_j)$ ;
9     end
10  end
11 end
12 Compute disjoint clauses  $\delta = \{\delta_1, \dots, \delta_k\}$  with  $\delta_i = S_i \wedge \psi_i$  // Here we suppose
    without loss of generality, the correspondences  $S_i$  and  $\psi_i$ 

```

4.3.1.3 Complete Artifacts

Definition 4.11. A *Complete Artifact* is a multi-sorted graph in which:

1. every applicable rule at an open node is applied and its execution simulated,
2. all constraints posed on the environment by each of the rules are captured.

Definition 4.12. (*Nodes of a complete artifact*)

A *task-node* (Q_t), represented by an ellipse, is a set of interdependent tasks – with dependent inherited and/or synthesized attributes –, such that the only missing attributes span from the environment (context). We denote by $\text{Inp}(Q_t)$, the set of input variables appearing in attributes inherited from the environment.

A **rules-to-apply-node** (Q_{ra}) or **r2a-node**, represented by a rectangle, is a set of business rules applicable at an open node associated with one of the tasks in a task-node, obtained after filtering based on a certain hypothesis h on data from the environment.

A **rule-node** (Q_r), represented by a circle, contains a single business rule selected from a r2a-node and corresponding to a user selection of a rule to apply at an open node.

Definition 4.13. Transitions of a complete artifact

Given a task-node Q_t and a hypothesis h on variables in $\text{Inp}(Q_t)$ based on data from the environment, that enables a r2a-node Q_{ra} , an **h-transition** (Figure 4.1) is an edge from Q_t to Q_{ra} labelled with the hypothesis h .

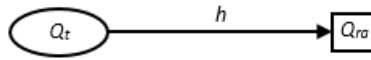


Figure 4.1 – An h -transition between a task-node and a r2a-node

Given a r2a-node Q_{ra} and a rule-node Q_r , an **r-transition** (Figure 4.2) is an edge between the two labelled by the selected business rule and the corresponding substitution σ .

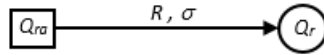


Figure 4.2 – An r -transition between a r2a-node and a rule-node

An **e-transition** (Figure 4.3) connects a rule-node to a task-node and an **ϵ -transition** (Figure 4.4) connects two task-nodes.

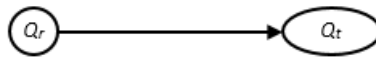
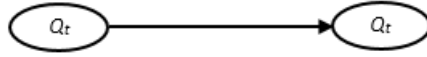


Figure 4.3 – An e -transition between a rule-node and a task-node

4.3.2 Building a Complete-Artifact

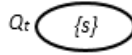
A complete artifact is a derivation tree of a guarded attribute grammar on which all possible executions of the grammar (starting from its axiom) are represented. It is constructed by iteratively simulating the operational semantics of guarded attribute grammars, distinguishing the node and/or transition types based on the two decision points of the GAG semantics (see below) after which they are produced. The following are the two decision points in GAG semantics:

Figure 4.4 – An ϵ -transition between two task-nodes

1. **Automated** selection of rules applicable at an open node,
2. **Manual** selection of a rule to apply at an open node.

The steps below describe the construction process

1. Create a task-node $Q_t = \{s\}$

Figure 4.5 – An initial task-node Q_t

2. Evaluate the guards of all rules with sorts in Q_t to extract the initial set of hypotheses $H = \{h_1, \dots, h_n\}$ on values from the environment.

If no rule exists for any of the sorts in Q_t , terminate the simulation of the branch and go to Step 9.

3. Using Algorithm 1, compute the discriminative clauses δ_i from H such that:
 - (a) $\delta_i \cap \delta_k = \emptyset$ for $i \neq k$ (pair-wise disjoint)
 - (b) δ_i is maximal.

Let $rules_i$ denote the set of business rules enabled by the clause δ_i .

4. For each of the couples $(\delta_i, rules_i)$ create a r2a-node $Q_{ra,i} = \{rules_i\}$ and add an h-transition labelled δ_i between Q_t and $Q_{ra,i}$.

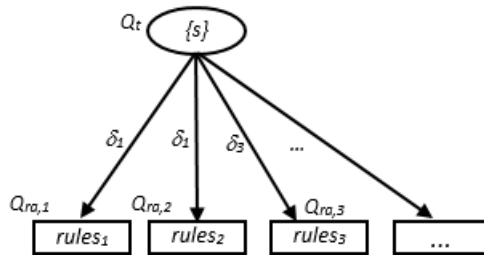


Figure 4.6 – Adding h-transitions and the list of enabled rules to the complete artifact

5. From each of the packet of rules $rules_i$, create rule-node subnodes for each of the rules in $rules_i$ and add r-transitions between $Q_{ra,i}$ and each of the new nodes. For instance if $rules_i = \{R1, R2, R3\}$, add the nodes as shown in Figure 4.7.

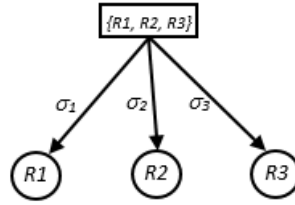


Figure 4.7 – rule-nodes and r-transitions

6. For each of the rule-nodes (R_i) added in the previous step,
 - (a) if $RHS(R_i) \neq \emptyset$, create new task-nodes (Q'_j) from the set $T = RHS(R_i) \cup (Q_t \setminus s)$, such that the task-nodes form connected components of the set T .

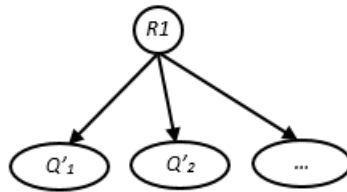


Figure 4.8 – Creation of new task-nodes

- (b) if $RHS(R_i) = \emptyset$, terminate the evaluation of the branch and go to Step 9.
7. If any of the newly created task-nodes is equivalent to a previous task-node in the complete artifact, stop processing the task-node and add an ϵ -transition from the new node to its old equivalent.

The rationale in this step is that the resolution of two equivalent nodes will always proceed in the exact same way (given that we exhibit all possibilities). Also, such situations occur because of recursive service calls. If the GAG is well-formed, that is, its termination is structurally guaranteed, such loops can be isolated and trimmed from the complete artifact tree.
8. For all the other task-nodes, restart the process from Step 2.
9. End.

4.3.3 Checking Soundness of a Complete Artifact

Based on the GAG operational semantics, for a task-node or a rule-node to be sound, all their sub-trees must be sound whereas for a r2A-node, it is required that only at least one of its sub-trees be sound. A leaf task-node corresponds to work for which no resolution method has been defined and denotes a configuration which can never be closed, hence not sound. In the following section, we present the steps to mark the nodes of a complete artifact, beginning from the leaves, with `Pass` or `Fail` depending on whether they are sound or not.

4.3.3.1 Marking a Complete Artifact

1. Start from the leaf nodes and mark each as follows:
 - **Fail** if it is a task-node
 - **Pass** if it is a rule-node
2. Progressively move up the tree and mark the other nodes as follows:
 - task-node: mark as **Pass** if all its subnodes are marked **Pass**, else, mark as **Fail**.
 - r2a-node: mark as **Pass** if at least one of its subnodes is marked **Pass**, else mark as **Fail**.
 - rule-node: mark as **Pass** if all its subnodes are marked **Pass**, else, mark as **Fail**.

Definition 4.14. (*Soundness of a non-recursive GAG*). A non-recursive autonomous guarded attribute grammar is said to be sound if and only if the root task-node of its complete artifact is marked **Pass**.

Note that for GAGs that possess recursive business rules, we cannot formally show that they eventually attain a terminal configuration. However, the non-deterministic nature of the GAG operational semantics gives users the ability to manually leave loops. Hence, in the remainder of this document, we describe GAGs that include recursion.

Conclusion

In Part II, we presented our attribute grammar-based model for task and dynamic process modelling, named **Active-Workspaces based on Guarded Attribute Grammars** (AW-GAG). The model uses hierarchical task decomposition paradigm of task analysis in which each task T is broken down into smaller chunks of work (subtasks) T_i and T is resolved by resolving all T_i .

Intuitively, the idea behind the AW-GAG model which makes it suitable to model dynamic user-centric and data-driven processes can be summarized in the following 10 points:

1. A user's workspace is made up of maps (artifacts) with open and closed nodes corresponding respectively to pending and resolved tasks.
2. Using hierarchical task analysis, a task can be decomposed in several ways, corresponding to different ways of resolving the task.
3. Each of these decompositions is captured in a grammar **production**, also called a **business rule** or simply **rule**.
4. Attributes (**inherited and synthesized**) and **semantic rules** are added to each production to implement the data flow and dependences between a task and its subtasks.
5. (Inherited) attributes are modelled as terms (patterns) which serve as **guards** that are matched against data from the context when filtering applicable business rules at open nodes. When pattern matching succeeds, it produces a substitution that defines the values of variables appearing at input positions of business rules.
6. If several rules are enabled after filtering at an open node, the choice of which to use is relegated to the user. Applying a business rule at an open node closes the node and creates new open nodes each for the subtasks of the rule.
7. The business rules for a task are independent hence new rules can be added on-the-fly without prejudice to existing ones.
8. The business rules for task and its subtasks form a **guarded attribute grammar** which can be considered as a **service**. Each service can be assigned to several users and each user can offer several services. In both cases, a copy of the GAG is placed in the so-called

user workspace of each of the users. The term **Active-Workspaces** is used to emphasize on the flexibility and control the user possesses over process design and enactment. Operationally, Active Workspaces capture the execution of a case (the manipulated data, the user decisions, etc.) from its inception to its completion in the so-called **artifacts** (mind-maps).

9. A user can call a service offered by another user by including in a business rule, a subtask for which no defining business rule exists in the local grammar. Such subtasks correspond to terminal symbols of the GAG and are viewed as references to GAGs in remote locations.
10. All communication accompanying service calls are exclusive through asynchronous exchange of messages.

In a nutshell active workspaces and guarded attribute grammars provide a modular, declarative, user-centric, data-driven, distributed and reconfigurable model of case management. It favours flexible design and execution of business process since it possesses (to varying degrees) all four forms of Process Flexibility proposed in [114]. We gave an in-depth description of this model through its syntax and its behaviour in Chapter 3. We paid attention to a crucial property of this model, input-enabledness, for GAGs that satisfy a monotony property. This property allows to distribute the model on an asynchronous architecture (Chapter 4). Input-enabledness is undecidable but we have identified the sufficient condition of strongly-acyclicity that can be checked very efficiently by a fixpoint computation of an over-approximation of attribute dependencies.

In Section 4.2, we describe a procedure to check the soundness of GAGs. Soundness is a property that asserts that any case introduced in the system can reach completion. This property is also undecidable [11] but we have defined a class of autonomous GAGs void of any loops on which we can check for soundness.

Part III

THE ACTIVE WORKSPACE FRAMEWORK

Introduction

To describe the modelling of a distributed collaborative system using GAGs, we proceed in two stages. First, we represent the workspaces of the various stakeholders as a collection of artifacts for the cases they handle. An artifact is a structured document with some active or live parts corresponding to pending work that may require the intervention of a user for their processing. Indeed, these active parts are each associated with a task that implicitly describes the data to be further substituted to the corresponding active part. For that reason, these workspaces are termed *active workspaces*. Second, we define collaborative systems by making the various active workspaces communicate and interact asynchronously through the exchange of messages.

This notion of *active workspaces* or *active documents* is close to the model of Active XML introduced by Abiteboul et al. [2] which consists of semi-structured documents with embedded service calls. Such an embedded service call is a query on another document, triggered when a corresponding guard is satisfied. The model of active documents can be distributed over a network of machines [1, 49]. This setting can be instantiated in many ways, according to the formalism used for specifying the guards, the query language, and the class of documents. The model of guarded attribute grammars is close to this general schema with some differences: First of all, guards in GAGs apply to the attributes of a single node while guards in AXML are properties that can be checked on a complete document. The invocation of a service in AXML creates a temporary document (called the workspace) that is removed from the document when the service call returns. In GAGs, a rule applied to solve a task adds new children to the node, and all computations performed for a task are preserved in the artifact. This provides a kind of monotony to artifacts, a useful property for verification purposes.

In this section, we describe the working environment of a user, how it is structured, the available tools and how users use them to drive process design, enactment and orchestration.

From the user's perspective, the objective of our work is to provide tools that boost his expressive power in process modelling and execution. In [79], we show that in the disease surveillance and outbreak management process, such an objective can only be attained if the postulant model provides support for the following four key aspects:

1. Iterative on-the-fly Process Design and Flexible Enactment:

The ability to iteratively build modular process models and to do so on-the-fly (at run-time). This coupled with increased user control of process enactment. By process enactment, we mean process execution viewed from a single user's perspective.

2. User Interactions and Process Orchestration:

Support different forms of user interactions, with an asynchronous (non-blocking) communication mechanism to enhance process orchestration. Process orchestration is process execution with a view on all users' workspaces.

3. Uncertainty and Exceptions:

Identifying all sources of doubt that might delay or prevent (user) actions and designing coping strategies for them.

4. Decision Making Support:

Providing users with enough information for easy decision making both at design- and run-time.

Chapter 5

Active Workspaces for User-Centered, Distributed Collaborative Systems

As we noted in the introduction, users play a central role in dynamic process and knowledge intensive process modelling. The objective of modelling these processes is to aid the user drive how process execution unfolds and not the other way round. Our model of collaborative systems is centered on the notion of user's workspace– the *active workspace*. This chapter is dedicated to the presentation of the Active-Workspace model, with emphasis on tools for the four key aspects discussed above.

5.1 The Active Workspaces Model

An active workspace groups together tools required for user-centered and data-driven process design and enactment. It provides support for all forms of work – automated work, manual work, and adhoc work –, facilitates interactions between users, and on-the-fly process design.

At the base, an active workspace contains a guarded attribute grammar specification of a process and several concrete instantiations created each time the process is invoked. Visually, a user's workspace is a collection of arborescent mindmap-like (simply called map hereafter) structures, – the so called artifacts –, that hold all the information about activities (tasks) the user carries out as well as the data required for and/or produced by these activities. Each of these maps is associated with a particular *service* offered by the user and each instantiation of the services creates a sub-tree at the root of the map.

For instance, the map shown in Fig. 5.1 might be found in the workspace of a clinician acting in the context of a disease surveillance system. Suppose that the clinician offers just a single service. His workspace therefore has a single map.

Figure 5.1 – Active workspace of a clinician

The service provided by the clinician is to identify the symptoms of influenza in a patient. To do so, we see from the first service instance on the map that he clinically examines the patient, administers some initial care, and eventually declares the patient as a suspect case.

Each call to this service, namely when a new patient comes to the clinician, creates a new tree rooted at the central node of the map. This tree is an *artifact* that represents a structured document for recording information about the patient throughout the consultation and treatment process. Initially the artifact is reduced to a single (open) node that bears information about the name, age and sex of the patient. An open node, graphically identified by a question mark, represents a *pending task* that requires the attention of the clinician.

5.1.1 Services and Roles

Services correspond to the axioms of a guarded attribute grammar. They are the entry points into the modelled process. In the example in Figure 5.1 only the consultation service is visible to the outside. It is the single point of entry into the clinician's workspace. Each active-workspace is associated with at least one service rendered by the user. We characterize a user's workspace by the services he offers and hence by the guarded attribute grammar specification of these services. As stated in Def. 3.3, (i) service sorts correspond to non-terminal symbols in guarded attribute grammars of distant workspaces, and (ii) each axiom is unique and does not appear in the right-hand side of any other business rule in its refining grammar.

Usually several users play the same **role** in a system. For example, in disease surveillance, there most likely exist several clinicians, several biologists, several epidemiologists, etc. This means that these users (in the same role) offer the same services and hence are attached to the same guarded attribute grammars (approximate to a renaming of the local sorts).

Formally, a role is defined by a generic GAG G and we obtain the disjoint union of these grammars as follows

$$\oplus(r :: R)G = \bigsqcup_{r :: R} G[r]$$

where r is a user who plays role R and $G[r]$ is the grammar obtained from G by replacing each sort (including the axiom s_0) by $s[r]$. Hence $s_0[r]$ represents service s_0 offered by r .

We note $G'\{G[r] \text{ where } r :: R\}$ the grammar made up of $\oplus(r :: R)G$ and of a grammar G' that calls this role. That is, G' will at some point need to request service G from a user in role R . This means that, in G' , we might find business rules with parameters such as $P[r :: R] : s \leftarrow \dots s_0[r_1] \dots$, expressing that when the user chooses rule P to apply at an open node, he inputs a user playing role R to whom the service request should be sent. The actual business rules are thus instantiations of these generic rules. We can also find in G' rules of the form $P : s \leftarrow s_0[r_1]$ expressing that a service call is made to all or some of the users in role R . In the latter case, the calling user still has to indicate the users who will receive the service calls. We note however the absence of parameters for P since the request will not be made to a particular user.

For simplicity, we equate the notions of role and service. Users who offer the same service are placed in a role for that service. For instance, if a system has two services S_1 and S_2 , and three users u_1, u_2, u_3 , with users u_1 and u_2 offering S_1 and users u_2 and u_3 offering S_2 , we create two roles R_{s_1} and R_{s_2} for S_1 and S_2 respectively such that $R_{s_1} = \{u_1, u_2\}$ and $R_{s_2} = \{u_2, u_3\}$.

When within a GAG G , calls are made to several external services (with GAGs G_1, G_2, \dots), we note

$$G\{G_1[r_1] \text{ where } r_1 :: R_1; G_2[r_2] \text{ where } r_2 :: R_2; \dots\}$$

or

$$G\{\oplus(r_1 :: R_1)G_1; \oplus(r_2 :: R_2)G_2; \dots\}$$

and this construction can be applied hierarchically to model chained calls as follows:

$$G_1\{G_2[r_2] \text{ where } r_2 :: R_2 \text{ and } G_2 = G\{G_3[r_3] \text{ where } r_3 :: R_3 \text{ and } G_3 = G\{\dots\}\}\}.$$

This hierarchical construction describes the complete collaboration scheme in the modelled process. As we will see later on, the process specification can be changed on-the-fly and users can be added or removed from roles dynamically. This chaining is thus dynamically updated to reflect the actual state of the process specification.

To conclude this section on services and roles, we note two important points:

- A service is offered by at least one user. Services for which no user has been assigned are not accessible.
- Users can be added or removed dynamically from roles. Or better still, a user can *subscribe* and/or *un-subscribe* from a role at any moment. Adding a new user to a role poses no particular difficulty since it does not modify existing workspace specifications but only increases the number of possible instantiations $G[r]$ than can be made from the grammar G . However, removing a user from a role might become problematic if there exist in his

workspace artifacts for the corresponding service with pending tasks. We can in such a situation either forbid the user from unsubscribing from the corresponding role or transfer the pending artifacts to the workspace of another user in the same role. We discuss the artifact transfer procedure in Section 5.3.1

5.1.2 Flexible Process Enactment and Incremental Design

A dynamic process by definition requires substantial *flexibility* at both design and run-time. This means that (i) the process is built on-the-fly and (ii) its execution does not necessarily follow some predefined order. We have already seen in Section 3.3 how guarded attribute grammars enforce data-drivenness by using guards to filter and hence suggest an ordering of process enactment. In the following paragraphs, we describe how the model supports flexible process enactment and how it can be used to support design at run-time.

5.1.2.1 Flexibly Process Enactment

Let s be a task defined by a form F , and let $rules2Apply(s)$ be the set of applicable rules obtained after matching F with the forms of business rules in some grammar G .

Algorithm 2 presents the general algorithm to a resolve s . In line 6, several rules have been found to solve s and it is inherent on the user to choose which one to apply. Recall from GAG monotony (Proposition 4.2.3) that incoming data not does not prevent the application of enabled rules in $rules2Apply(s)$ and hence the only form of non-determinism in guarded attribute grammars correspond to the user's action in line 6. This non-determinism gives users considerable control over the enactment of the process and boosts the user-centeredness property of active-workspaces. In addition, in both cases of the IF-block between lines 3 and 8, before a rule is applied, the user might still need to intervene to provide additional input data and in the case where the refinement includes one or more service calls, indicate the distant user(s) to whom service requests should be sent.

Also, the non-existence of any apparent ordering on tasks as well as the production of values in push mode both contribute to enhance the flexible enactment of guarded attribute grammars. More precisely, a term in an inherited position need not be completely defined before the value is sent where it is used. In like manner, a task needs not wait for all its attributes to be completely defined before it starts its execution. Execution can therefore proceed with holes in the data. These holes are progressively filled by produced data values and they have as effect to either invalidate certain rules or authorize new rules at open nodes.

Algorithm 2: General task resolution algorithm

```

Data: Task  $s$ 
1 compute  $rules2Apply(s)$ ;
2 if  $|rules2Apply(s)| > 0$  then
3   if  $|rules2Apply(s)| = 1$  then
4     | refine the artifact with the single rule;
5   else
6     | let the user choose which to apply;
7     | refine the artifact with the selected rule;
8   end
9   restart the algorithm for each newly created subtask;
10 else
11 | Halt. Task cannot be resolved.;
12 end

```

5.1.2.2 Incremental Process Design

The active-workspaces model can be viewed as two-layered, the lower layer holding the guarded attribute grammar specification of a process, and the upper layer consisting of an execution engine that implements the operational semantics of guarded attribute grammars. The two layers are structurally independent since they can both evolve independently, and functionally dependent since the upper layer runs by interpreting the script in the lower layer. This means that the underlying grammar specification can be updated, and subsequent executions will use the updated specification. Indeed, the GAG specification is not stored in memory, line 1 of Algorithm 2 parses, compiles, and instantiates the rules in the GAG on each execution.

Also, business rules corresponding to different refinements of the same task are completely independent. They share the same sort and will eventually be brought together during task resolution, but on the specification script, they are completely independent entities. This means that new rules can be added to the script at any point in the lifecycle of the process. We can imagine an implementation of an active workspace system where by default the process specification is completely built on the fly. That is, the **else** statement in lines 10 and 11 of Algorithm 2 does not exist and the user is instead prompted to create a new rule when none is applicable or when none of the applicable ones suit the pursued goal. Adding new rules at runtime entails adding new data, tasks, users, and semantic rules. We call this feature of active workspaces, **late modelling**.

Late modelling involves late data modelling, late semantic-rule modelling, and might also include late user/role modelling when external service calls are made. One way to implement late modelling to design new rules in the Algorithm 2 could be: (i) generate the left-hand side of the new rule based on attributes of the form for the input task s , (ii) let the user define semantic rules on the right-hand side eventually with users and roles where necessary, and finally apply

the rule to resolve s .

This way of doing work is suited for expert activities as described in Section 2.5.2 where at any instant, a user can deviate from the prescribed procedure and model a new ad-hoc equivalent procedure in response to the needs on the field. Modelling a new adhoc procedure entails creating new tasks, assigning actors to them, and planning their execution.

5.1.3 Relationships between Tasks

Example 5.1.1.

$$\begin{aligned}
 R1: & \quad A(x)\langle z' \rangle \rightarrow B(x)\langle y \rangle C(y)\langle z \rangle D(y, z)\langle z' \rangle \\
 R2: & \quad B(x)\langle b(x) \rangle \rightarrow \\
 R3: & \quad B(x)\langle d(x) \rangle \rightarrow \\
 R4: & \quad C(b(x'))\langle z \rangle \rightarrow \dots \\
 R5: & \quad C(d(x'))\langle z \rangle \rightarrow \dots \\
 R6: & \quad D(y, z)\langle z \rangle \rightarrow \dots
 \end{aligned}$$

When rule $R1$ is applied at some open node of an artifact, it produces three new open nodes for tasks $B(x)\langle y \rangle$, $C(y)\langle z \rangle$ and $D(y, z)\langle z' \rangle$. In the following paragraphs, we use these newly created open nodes to describe relationships between tasks. *End of Example 5.1.1*

The only execution order on tasks is that based on their data dependencies. In the Active-Workspaces model, the graph of local dependences can be used to deduce an ordering in the execution of tasks. We use Example 5.1.1 to describe how different types of tasks dependencies are achieved in the GAG model.

5.1.3.1 Concurrency or Order Independence

Concurrency and Order-Independence task relationships are by default assumed during process enactment in the active-workspace model. This is supported by the confluence property of guarded attribute grammars (see Corollary 4.2.4).

In Example 5.1.1, task D proceeds concurrently with task B and rule $R6$ is applied. This happens because the rule $R6$ for D poses no constraints (guards) on the form of its inputs. The execution of D simple transfers the references of the variables y and z to its subtasks.

5.1.3.2 Sequence

Sequence relationships are induced between subtasks when one produces values used to guard the execution of the other. Task C in the above example will not be able to proceed since there will be no enabled rule at its open node. The rules for C , $R4$ and $R5$, are guarded by the value of y produced by B . $R4$ checks if the value of y is of the form $b(x')$ and $R5$ checks if the value of y is of the form $d(x')$. Task C therefore has to wait for B to complete before proceeding. This imposes an ordering in the execution of B and C and we say that they are in a **sequence** relationship.

There are three types of sequence relationships between two tasks A and B [6]: (i) meets: when B starts immediately after A finishes, (ii) precedence: when B starts after A finishes with a lead time t , and (iii) overlapping: when B can start once A has started. Overlapping is naturally supported by the operational semantics of GAGs. Note that the end of the processing of a task is not explicit in the GAG operational semantics. A task is considered resolved once a rule has been applied at its open node, and nothing is said about the effective end of the processing of the task, that is, when no open nodes exist in its artifact sub-tree. To enforce meet and precedence relationships, we use an explicit attribute defined by A and used by B whose value is only set when the processing of the task is completed. This value can be set automatically by scanning through the sub-tree to check that no open nodes exists or manually by the user.

5.1.3.3 Optionality

One other relationship that is achievable with guarded attribute grammars is **optionality**. When a task is optional, its defining business rule takes the following form

$$A \rightarrow () \mid A()\langle \rangle$$

denoting that a task of sort A can be skipped by applying an empty rule.

Optionality is not automatically enforced; the user has to manually select an empty rule at runtime for it to be achieved. If the task chosen as optional produced values that are used elsewhere, the user will be prompted to manually provide these values.

More precisely, if $s(\dots)\langle y_1, \dots, y_m \rangle$ is chosen to be optional in a configuration with substitution σ , then the user will be prompted to provide values for every y_j , $1 \leq j \leq m$ such that $y_j \in Var(\sigma)$.

5.1.3.4 Loops or Recursion

As seen earlier, the guarded attribute grammar model allows for loops specified by rules of the form

$$A \rightarrow \dots A \dots$$

which indicate a recursive call to task A . For every **iteration** relationship, there must exist a corresponding rule of the form

$$A \rightarrow^* \mathcal{D}, \text{ s.t. } A \notin \mathcal{D},$$

that is, whose right-hand side when completely refined does not contain a task of sort A . This rule serves as a guarantee that eventually the process will exit the loop – an important aspect of soundness check –.

5.2 Timed Guarded Attribute Grammars

Time is a critical and determining factor in user-satisfaction, cost reduction, process analysis, and productivity of business processes [124, 42]. In dynamic processes such as disease surveillance, timeliness is a major metric used to assert and/or evaluate the effectiveness and business relevance of modelled processes [123, 134, 7].

In the Active-Workspaces model, each active workspace is assigned a clock \mathcal{T} which registers the creation time of each of the nodes of its artifacts. We note $\mathcal{T}(n)$ the creation time of node n . In rule-based approaches to process modelling, we do not always have a complete view of the entire process. It is therefore not possible to have a completely timed process model at specification time. Instead, we reason about when each of the business rules can be applied.

In a *timed guarded attribute grammar*, each business rule is labelled with a time interval called its *validity interval*:

$$(t_{min}, t_{max})$$

expressing that the rule when enabled at an open node n cannot be applied before $\mathcal{T}(n) + t_{min}$ and after $\mathcal{T}(n) + t_{max}$. For rules that are immediate applicable, t_{min} is set at 0. Time is therefore used during enactment to progressively filter applicable rules at open nodes.

For example, in disease surveillance, open nodes for the data collection and data analysis tasks might be created at the same time. The data collection node starts immediately hence it should have at least a rule with $t_{min} = 0$ whereas the data analysis rules will need to wait sufficient time $t_{min} > 1$ to allow for enough data to be collected or for all data collectors to complete before they can be applied.

To avoid a situation where all applicable rules become filtered out with time, we add one rule of each sort whose validity interval is (t, ∞) with t larger than the t_{max} 's of all alternative rules for that sort. This ensures that all open nodes will eventually be closed given that when time $\mathcal{T}(n) + t$ is reached at an open node n , only this one rule will be enabled. Such a rule corresponds to an action that will automatically be executed by the system if all other rules fail to be enabled and chosen by the user within their validity intervals.

5.2.0.1 Periodic Tasks

The active workspaces model also provides support for tasks that reoccur indefinitely with a fixed periodicity. A periodic call to a task is preceded by the keyword **recur**[*period*] with a parameter denoting the period that separates two executions. The latter is expressed in the following time units: m (minutes), h (hours), d (days), and w (weeks). For instance, the following specification executes task s_i every 2 weeks:

$$s_0 \rightarrow \dots \mathbf{recur}[2w] s_i \dots$$

An equivalent specification uses a recursive call $s \rightarrow \dots s \dots$ with a timed rule for s whose validity interval is $[p, \infty]$ where p is the period of execution. If the latter is the only applicable rule for s , it will be automatically applied every p time units.

5.3 Collaboration and User Interactions

First, we describe the social environment around which the active workspace model is built. By social environment, we mean one characterized by a common world in which participating social entities interfere with, depend on, and influence each other's actions [16]. A social entity therefore has goals and might have to communicate and/or interact with other entities to attain its goals.

Users in the active-workspace model are social entities. They interact and communicate in myriad ways, each on a well-defined set of sub-goals necessary to attain a larger goal. We say that the users collaborate or cooperate. Cooperation in active-workspaces is strict, that is, an agreement is reached between the users and each of them is aware of the others intention to exploit them, as against loose cooperation in which a user can exploit the actions of another user without the latter knowing or consenting [16]. In our model, the reached agreement corresponds to the published services in each active workspace and each user is aware of the services he offers and of the external services he can reach.

Guarded attribute grammars provide effective support for collaborative or cooperative work. In Section 4.1, we describe a message exchange mechanism that enhances communication between non-autonomous guarded attribute grammars. Recall that the latter though geographically distant can be viewed as a (composed) global grammar whose behaviour is nothing short of the behaviours of the individual guarded attribute grammars put together. Also, in Section 4.2, we describe a property of guarded attribute grammars, namely monotony (see Proposition 4.2.3), that guarantees effective distribution by ensuring that artifacts only grow and that executed atomic steps cannot be subsequently revoked.

Collaboration is essentially by exchange of services. Generally speaking, each Active-Workspace can be seen to possess a work-basket into which service calls destined for it are placed. The work-basket also serves as a buffer to store variables and values exchanged in subscriptions. These services, variables, and values are exchanged as messages between Active-Workspaces.

In the following subsection, we describe two major forms of user interactions provided by the active-workspaces model to enhance collaboration between users: requesting or assigning work and transferring or migrating work.

5.3.1 Requesting or Assigning Work

When a user requests work from or assigns work to another user, we say that he delegates work to or relies on the latter. In simple terms, delegation can be seen as a situation where user r_1 needs or likes an action of user r_2 and includes it in his own plan [16]. Delegation is the basic ingredient for a plethora of interaction needs: help, exchange, partnerships, teamwork.

Work delegation is achieved by service calls in guarded attribute grammars. To initiate a service call, the caller indicates a user (the receiver), amongst those in the role for the service, to whom the call should be made. This effects the creation of a new artifact (reduced to a single open node) at the root of the mind-map for the corresponding service at the receiver's workspace.

Recall from Section 3.2 that service calls are modelled as terminal symbols appearing in a guarded attribute grammar. That is, they have no defining rule in the grammar, but make reference to another grammar defined in a distant active workspace. We denote the sorts of forms for service calls by $s[r :: R]$ expressing that at runtime, the service should be sent to a user r belonging to role R . An example of a form for a service call to a single user is given below:

$$s[r :: R](x_1, \dots, x_n)\langle y_1, \dots, y_m \rangle.$$

In Section 4.1 we describe the different types of messages how they are exchanged during a service call.

GAGs even add flexibility to work delegation by allowing for *one-to-many* or *multi-user* services calls. This means that in a given process, several users can be assigned the same subgoal(s). Multi-user service calls can be either *crowd-based* or *group-based*.

1. Crowd-based multi-user service calls

In a crowd-based service call, a user requests a service from one or more users and only the requesting user is aware of the multiplicity. In the case where it is sent to more than one user, each of them is unaware of the other and works as if he was the only one rendering the service.

A crowd-based service call is denoted by the keyword **crowd**, followed by a form whose sort is written as $s[R]$ on the right-hand side of a business rule. Such a sort is called a generic sort and the rule to which it belongs is a parametric rule. We discuss the latter in detail in Section 5.4.2.2.

$$\mathbf{crowd} \ s[R](x_1, \dots, x_n)\langle y_1, \dots, y_m \rangle.$$

The **crowd** keyword indicates that a value should be retained for each of the synthesised attributes from each of the users in R . Hence each y_j will be a vector indexed by the elements of R . Using vectors does not change the operational semantics of GAGs since a vector simply groups together variable subscriptions where a single local variable will contain several values. The entries of the vector can be viewed as distinct local variables.

We define a projection operation on such attributes, denoted $y[r]$, to extract the value of y synthesized at the workspace of user $r \in R$. However, in general we are not aware of the presence of a particular individual in a given role since users subscribe and/or unsubscribe from roles dynamically. We are therefore rather interested in aggregating the data in these vectors and posing constraints such as: "there exist $r \in R$ such that $y[r]$ satisfies \mathcal{C} " – \mathcal{C} being some condition on values of the vector y – or "for all $r \in R$, $y[r]$ satisfy \mathcal{C} ", or even "there exist at least 3 individuals $r \in R$ such that $y[r]$ [satisfy \mathcal{C}]", or if vector y holds Boolean values, "at least 50% of $r \in R$ verify $y[r]$ ". More generally one can express the semantic rules using any kind of functional expression as long as the values of inherited attributes evaluate to terms so that they can be matched against patterns.

In certain situations, it might be necessary to wait until a certain prerequisite on the number of responses from a crowd-based service call is reached before evaluating a condition on the values. For instance, if a service request is made to n persons to reply by Yes or No to a certain question, and if these responses are used in a guard, say 50% of $r \in R$ verify $y[r] == \text{Yes}$, it is reasonable to wait for a considerable number of responses to be received before evaluating the guard. Waiting is not naturally supported by the GAG semantics since guards for rules applicable at an open node are automatically evaluated when the node is created.

We therefore add a "when" operator (**when nb**) that specifies the number of responses that should have been received before the guards for a rule are evaluated. The parameter **nb** is expressed as a decile denoting the percentage of responses that should be received prior to evaluating the guard. The above guard therefore becomes: **50% of $r \in R$ verify ($y[r] == \text{Yes}$) when 60**, denoting that the guard can only be evaluated when 60% of the n awaiting responses would have been received.

Waiting as described above does not affect the monotony property of guards since the atomic step (rule application) is delayed until the guard is evaluable and the intermediary steps that generate a substitution for the rule is updated at each re-evaluation. More so, making a rule wait does not affect the evaluation of the guards of the other rules. In fact, if the user chooses one of the rules whose guards have already been evaluated successfully, every other rule (including those still waiting) is discarded and subsequent values received are discarded.

2. Group-based Service calls

Group-based service calls model users working together on an equal status to attain a common goal. This form of collaboration is a synchronous activity with no division of labour and no multiplicity in the responses. The users share a common workspace and the decisions made during process enactment are based on consensus reached by all or some of the participating users. An example is parliamentary deliberations where the parliamentarians sited in an auditorium discuss and deliberate together validating and/or rejecting submitted bills.

Unlike with individual user-workspaces, a group active-workspace is only accessible in connected mode. Groups are built from roles, that is, members of a group belong to the same role or can offer the same service. Hence a role R contains not only users but smaller groupings of users to whom group-based service calls can be made.

We denote a group-based service call by the keyword **group**, followed by a form with a generic sort $s[R]$ on the right-hand side of a business rule. For example,

$$\mathbf{group} \ s[R](x_1, \dots, x_n)\langle y_1, \dots, y_m \rangle$$

models a service call to a group and the enacting user is expected at runtime to either constitute a new grouping of users in R or select a pre-existing one to whom the service call will be made. The semantics of group-based service calls is equivalent to that of single user service calls but for the fact that decisions which were made by a single user at enactment now have to be made by a group of users.

The group active workspace is hence equipped with tools to aid in making decisions during process enactment. The group is charged with choosing the decision-making strategy to adopt when handling a case. Such strategies could include: (i) assigning administrator privileges to some members of the group and allowing the other members to give views

and opinions that guide the administrator(s) make decisions at runtime, (ii) collectively reaching a consensus by way of vote or polls. Examples of the latter could be majority voting rule, weighted voting, or probabilistic voting [94]. The detailed study of such decision-making strategies is beyond the scope of this thesis.

We define the following macros on top of the GAG model for the different ways of initiating service calls. Macros 1 and 5 allow for optionality since **none** is an acceptable value for R while Macros 1, 2, and 3 are used in the context of crowd-based and group-based multi-user service calls.

1. *** Many**: denoted $s^*[R](\dots)\langle\dots\rangle$ for a service call made to none or several users in R .
2. **+ Some**: denoted $s^+[R](\dots)\langle\dots\rangle$ for a service call made at least one user in R .
3. **~ All**: denoted $s^\sim[R](\dots)\langle\dots\rangle$ for a service call made to all users in R .
4. **! One**: denoted $s^![R](\dots)\langle\dots\rangle$ or simply $s[r :: R](\dots)\langle\dots\rangle$ for a service call made to exactly one user in R .
5. **? To**: denoted $s^?[R](\dots)\langle\dots\rangle$ for a service call made to none or exactly one user in R .
6. **# User**: denoted $s^\#[user_1, user_2, \dots](\dots)\langle\dots\rangle$ for fairly stable systems. It is possible to directly specify in the grammar a user or users to whom a service call should be sent at run time.

5.3.2 Artifact Sync and Migration

A user to whom a service call has been made might become unavailable to complete the work, or it might be required to remove a user from a role with open artifacts for the corresponding service in his active workspace. In both cases it is important to preserve the work already done. The **artifact migration** or **artifact transfer** feature is used to move artifacts between workspaces in the same role.

First, given that two users offering the same service might not have the same guarded attribute grammar specification because they can evolve their workspaces independently, it might be necessary to synchronize the grammar specifications between two sites before proceeding with the transfer of an artifact.

We denote by $sync(G_l, G_d)$, the **synchronization** operation which updates the local GAG G_l using rules from the distant GAG G_d , with $axiom(G_l) = axiom(G_d)$. It is defined as follows:

$$sync(G_l, G_d) = G_l \cup \{R \mid R \in G_d, R \not\equiv R', R' \in G_l \text{ and } sort(R) = sort(R')\}$$

with $R \not\equiv R'$ defined as follows:

- $|A_{inh}(R)| \neq |A_{inh}(R')|$ or $|A_{syn}(R)| \neq |A_{syn}(R')|$
- $\forall t_i, t'_i, \quad t_i \in A_{inh}(R)$ and $t'_i \in A_{inh}(R'), \quad t_i \not\equiv t'_i, \quad 1 \leq i \leq |A_{inh}(R)|$
- $\forall u_i, u'_i, \quad u_i \in A_{syn}(R)$ and $u'_i \in A_{syn}(R'), \quad u_i \not\equiv u'_i, \quad 1 \leq i \leq |A_{syn}(R)|$
- $|rhs(R)| \neq |rhs(R')|$
- $\forall s, s', \quad s \in sorts(rhs(R)), s' \in sorts(rhs(R')), \quad s = s'$
 - with $s = (t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ and $s = (t'_1, \dots, t'_q)\langle y'_1, \dots, y'_p \rangle$
- $m \neq q$ or $m \neq p$
- $\forall i \in [1, n] \quad t_i \not\equiv t'_i$

The sync operation preserves the properties of the initial grammars. It is simply a disjoint union of the rules in G_l with those in G_d . Note that $t \not\equiv t'$ correspond to a pattern matching that fails.

Now we define the operation of transferring an artifact with configuration Γ from a distant active-workspace with GAG G_d to a local active-workspace with GAG G_l . Let M be the set of messages generated by Γ . M is divided into an input buffer and an output buffer.

To achieve this we add two additional types of messages:

- **artifact-transfer**: sent from d to l , made up of the configuration of the artifact Γ and the list of messages M emanating from Γ .
- **rename**: sent from active-workspace l to every other active-workspace referenced in any of the messages in M .

The following steps, are executed to effect the transfer:

1. Obtain Γ and M at active-workspace d ,
2. Send an **artifact-transfer** message from d to l with the contents Γ and M .
3. When an **artifact-transfer** message is received at active-workspace l , it renames all nodes and variables in Γ , from the namespace $ns(d)$ to the namespace $ns(l)$, and generates a set of new names whose elements are couples of the form $\{(oldName, newName)\}$.
4. For each couple $(oldName, newName)$, if $oldName$ appears in a message $m \in M$, then:
 - (a) Rename $oldName$ with $newName$
 - (b) Obtain the distant workspace appearing in m , say d' , and send a **rename** message containing the old and new names to d' .
5. When a workspace receives a **rename** message, it scans its list of messages for occurrences of each $oldName$ and replaces them with the corresponding $newName$.

5.4 Towards a Language for Active-Workspace Specification

In this chapter, we lay the groundwork for the development of a declarative scripting language for guarded attribute grammars. We start by augmenting the GAG syntax with predicate logic formulae and functional expressions that respectively enrich the guards and allow for automated computation in semantic rules. Then we present a language-oriented syntax and use it to model snippets of the disease surveillance process.

5.4.1 Enriching Guards and Semantic Rules

5.4.1.1 Using Predicate Logic in Guards

Filtering (checking the applicability of) a business rule at an open node proceeds in two phases: (i) check the presence of the inherited attribute values in the task definition, and that these values match with the patterns at the corresponding inherited positions of candidate business rules, and (ii) define a substitution σ (σ_{in} precisely) which initialises the local variables of the business rule with values from the task definition. It also checks that there exists a synthesized attribute in the rule that serves as a promise for a value for every output variable of the task. For simplicity, we do not consider this last part.

Filtering using pattern matching alone is not enough to model real world situations. For instance, consider the following definition of the left-hand side of a rule of sort patient_visit.

$$\underline{\text{patient_visit}}(\text{name}, \underline{\text{dob}}(\text{day}, \text{month}, 1980), \text{gender})\langle \dots \rangle$$

Given a task t , the filtering algorithm checks that t has three input values, and that the second value matches the pattern dob(day, month, 1980), that is, it checks that the value constructor is dob, that it has three components, and the third component is **equal to** the constant constructor 1980. If

$$t = \text{patient_visit}(\text{"Lee"}, \underline{\text{dob}}(12, 05, 1980), \text{"Male"})\langle \dots \rangle,$$

the filtering will succeed with the substitution,

$$\sigma = \{\text{name} = \text{"Lee"}, \text{day} = 12, \text{month} = 05, \text{gender} = \text{"Male"}\}.$$

Pattern matching was only able to check the **equality** of values. In a practical situation, we

will want to define such a business rule for patients in an age range. More so, placing constant values directly in patterns as such makes them unusable in the right-hand side of the rule since there exist no recipient variable for their values. Notice the absence of `year` in σ .

We augment the syntax rule syntax with predicate logic formulae using a **where** clause as follows:

$$\mathbf{rule_sort}(x_1, \dots, x_n)\langle y_1, \dots, y_m \rangle \mathbf{where} [cond] \rightarrow RHS$$

with $[cond]$ being a comma-separated list of predicate logic formulae on the variables occurring in inherited attributes (x_i) of the business rule –*the variables that will be initialised in $Var(\sigma_{in})$* .

$$cond :: Var(\sigma_{in}) \rightarrow \dots \rightarrow Var(\sigma_{in}) \rightarrow Boolean.$$

We can modify the business rule above to be used for male patients born between 1975 and 1985.

patient_visit (name, dob (day, month, year), gender)⟨...⟩
where [(year > 1975 \wedge year < 1985), gender = "Male"],

Filtering this rule against t produces

$$\sigma = \{\text{name} = \text{"bob"}, \text{day} = 12, \text{month} = 05, \text{year} = 1980, \text{gender} = \text{"Male"}\},$$

key/value pairs on which the conditions are evaluated, and the associated business rule only becomes enabled at an open node if all the conditions evaluate to True.

The **where** clause is also used to check conditions on value vectors produced by crowd-based service calls. For that purpose, we define several macros for computing certain aggregations of values in the vectors and eventually testing conditions on them.

5.4.1.2 Extending Semantic Rules with External Features

In practical terms, a business process is an organisational model in which individual autonomous systems inter-operate. For instance, the data in a business process will need to be stored in a database or some tasks might need to run on data in external databases, certain automated tasks might need specialised algorithms which have to be implemented in some external environment, etc. We identify three major reasons to motivate the coupling of semantic rules with external features:

1. **Support automated computations:** simple data manipulation computations always come in handy in process execution. For instance, basic arithmetic operations (addition,

difference, multiplication, division), computing simple descriptive analysis (sum, mean, median, mode) on a list of values, basic arithmetic operations, basic string manipulation operations (prefix, suffix, substring, ...), etc. Such operations should be usable directly in the process definition or encapsulated in a function, for example, a function could be used to compute a person's age from his date of birth. These functions are, in any case, necessary to describe the various plumbing operations between the a GAG specification and its context of use. They are lazily evaluated and hence do not really affect the operational semantics of GAG.

2. **Access to external systems:** examples of external systems include, platforms for different departments of an institution: human resources, customer relations, logistics, finance and control, etc. or general-purpose platforms for data management and analysis, database or data-warehouse management, messaging, etc.
3. **Extracting information from enacted artifacts:** artifacts store all the information manipulated throughout the lifecycle of case. Such information can be useful to build dashboards or to feed some local databases that are later used to:
 - (a) respond to specific requests from other users, for example in the context of a checklist (see Section 5.4.2.3). They are also used to populate some input parameters (see Section 5.4.2.2) of a rule in place of the user.
 - (b) guide the user to choose a rule to apply for a pending task,
 - (c) used cohesively with data mining and decision support techniques to suggest a specific rule to apply, prioritize enabled rules at an open node, or even inhibit some of the rules.

Such implicit side-effects of rules abound, and they generally do not conflict with the GAG specification but rather complement it since they allow to associate real-world activities with a rule. We achieve such side effects by augmenting the active-workspace model to use web-services and database drivers.

We note that resorting to external databases and web-services will have to be done in a way that preserves the monotony of the specification and hence guarantees its safe distribution. Similarly, we must be careful, if side-effects can inhibit some rules, that this does not jeopardize the soundness of the GAG. By the way, it is important to dispose of a language to describe side-effects of rules and in particular a language for making queries on active workspaces. The ideal solution would be to implement GAG as a domain specific language embedded into a general-purpose language. In that case we could directly write the side-effects in the host language. In Chapter 6 we implement one such internal DSL, embedded into Haskell, for guarded attribute grammar specification and execution.

Formally, we use the following syntax to denote a call to an external feature:

$$\mathbf{f} \text{ function_name } [arg1 \ arg2 \ \dots]$$

The \mathbf{f} indicates a call to an external function or access to an external feature, and is followed by the name of the function, and a possibly empty list of arguments. For instance, the following is a call to a function that computes the age from the year of birth:

$$\mathbf{f} \text{ age year}$$

5.4.2 An Extended Syntax for GAGs

In this section we introduce some syntax elements to outline a specification language for guarded attribute grammars that is expressive enough to describe realistic applications. Our purpose is not to fully design such a specification language. In Chapter 6, we present the design of a prototype specification language inspired by the syntax in this section, which supports distributed execution and provides an implementation of a typing mechanism for the manipulated values and guards. This section only intends to introduce some syntactic sugar and constructs which allow to describe large and complex specifications in a more concise and friendly way.

First, we introduce a functional notation for business rules, inspired from monadic programming in Haskell. So far, a guarded attribute grammar was presented as a task rewriting system, a convenient formalism for formal manipulations. However, rewriting systems are not necessarily perceived as a handy programming notation despite their similarity with logic programming. In Section 5.4.2.2 we describe a mechanism for writing generic rules, that is, rules that contain parameters that when instantiated can generate a potentially large set of similar rules. As we showed in Chapter 5, parametric rules are particularly useful to formalize the notion of role: When several stakeholders play a similar role, they can use the same generic local grammar instantiated with their respective identities to distinguish them from one another. Section 5.4.2.3 introduces a feature that allows the designer to extend the formalism by adding combinators, a technique that can be used to customize the notations in order to derive domain specific languages adapted to the particular user needs.

5.4.2.1 A Functional Notation

We rewrite the business rule

$$\begin{aligned} \text{sort}(p_1, \dots, p_n) \langle u_1, \dots, u_m \rangle \mathbf{where} [cond] \rightarrow \\ \text{sort}_1(t_1^{(1)}, \dots, t_{n_1}^{(1)}) \langle y_1^{(1)}, \dots, y_{m_1}^{(1)} \rangle \\ \dots \\ \text{sort}_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \langle y_1^{(k)}, \dots, y_{m_k}^{(k)} \rangle \end{aligned}$$

as

$$\begin{aligned} \text{sort}(p_1, \dots, p_n) \mathbf{where} [cond] = \\ \mathbf{do} (y_1^{(1)}, \dots, y_{m_1}^{(1)}) \leftarrow \text{sort}_1(t_1^{(1)}, \dots, t_{n_1}^{(1)}) \\ \dots \\ (y_1^{(k)}, \dots, y_{m_k}^{(k)}) \leftarrow \text{sort}_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \\ \mathbf{return} (u_1, \dots, u_m) \end{aligned}$$

In this new notation, $\text{sort}(p_1, \dots, p_n)$ represents the rule *signature*, the lines between the keywords **do** and **return**, $(y_i^{(k)}, \dots, y_{m_i}^{(k)}) \leftarrow \text{sort}_i(t_1^{(k)}, \dots, t_{n_i}^{(k)})$, represent generators. Variables subscribing to the generated values –or terms built on one or more of these values–, are placed in the statement: **return** (u_1, \dots, u_m) . For example, we rewrite the *patient_visit* rule using the do-notation as follows:

$$\begin{aligned} \text{patient_visit} (\text{name}, \underline{\text{dob}} (\text{day}, \text{month}, \text{year}), \text{gender}) \\ \mathbf{where} [(\text{year} > 1975 \wedge \text{year} < 1985), \text{gender} = \text{"Male"}] = \\ \mathbf{do} (\text{symps}) \leftarrow \text{clinicalAssessment} (\text{name}, \text{gender}, \text{year}) \\ (\text{rep}) \leftarrow \text{initialCare} (\text{symps}) \\ (\text{ack}) \leftarrow \text{caseDeclaration} (\underline{\text{patient}} (\text{name}, \text{gender}, \text{year}), \text{symps}) \\ \mathbf{return} (\underline{\text{report}} (\text{symps}, \text{rep}), \text{ack}) \end{aligned}$$

This functional presentation stresses out the operational purpose of business rules: Each task has an input –inherited attributes– seen as parameters and an output –synthesized attributes– seen as returned values. This notation however can confuse Haskell programmers for two reasons.

First, recall from Definition 3.3 that an input occurrence of a variable is either a variable occurring in a pattern p_i or a variable occurring as a subscription in the right-hand side of the rule or, in this alternative presentation, in the left-hand side of a *generator*

$$(y_1^{(j)}, \dots, y_{m_j}^{(j)}) \leftarrow \text{sort}_j(t_1^{(j)}, \dots, t_{n_j}^{(j)})$$

Note that, using this **do** notation, a guarded attribute grammar is left-attributed (Def. 4.4)

precisely when every variable is defined before used: all output occurrences of a variable are preceded by its corresponding input occurrence. For guarded attribute grammars which are not left-attributed we thus find some variables which are used before being defined. This is incompatible with monadic programming in Haskell where the scope of a variable occurring in the left-hand side of a generator is the part of the **do** expression that follows the generator, including the return statement.

Second, a Haskell monadic expression is evaluated in *pull* mode, that is, if the output returned by the **do** expression does not use the values of the variables defined by a given generator, then this generator is not evaluated at all. By contrast, a rule of a guarded attribute grammar is evaluated in *push* mode: When rule is applied, we create one open node for every generator. Then users can continue to develop these nodes with the effect of gradually refining the returned values.

Example 5.4.1. Consider the GAG of Example 3.2.1:

$$\begin{aligned} \text{Root} & : \quad \text{root}() \langle x \rangle \textbf{where} \quad [] \rightarrow \text{bin}(\text{Nil}) \langle x \rangle \\ \text{Fork} & : \quad \text{bin}(x) \langle y \rangle \textbf{where} \quad [] \rightarrow \text{bin}(z) \langle y \rangle \text{bin}(x) \langle z \rangle \\ \text{Leaf}_a & : \quad \text{bin}(x) \langle \text{Cons}_a(x) \rangle \textbf{where} \quad [] \rightarrow \end{aligned}$$

Its syntactical translation into the functional notation is as follows:

$$\begin{aligned} \text{Root} & : \quad \text{root}() \textbf{where} \quad [] = \textbf{do} \quad (x) \leftarrow \text{bin}(\text{Nil}) \\ & \quad \quad \quad \textbf{return} \quad (x) \\ \text{Fork} & : \quad \text{bin}(x) \textbf{where} \quad [] = \textbf{do} \quad (z) \leftarrow \text{bin}(x) \\ & \quad \quad \quad (y) \leftarrow \text{bin}(z) \\ & \quad \quad \quad \textbf{return} \quad (y) \\ \text{Leaf}_a & : \quad \text{bin}(x) \textbf{where} \quad [] = \textbf{do} \quad \textbf{return} \quad (\text{Cons}_a(x)) \end{aligned}$$

which we write simply as

$$\begin{aligned} \text{Root} & : \quad \text{root}() = \text{bin}(\text{Nil}) \\ \text{Fork} & : \quad \text{bin}(x) = \textbf{do} \quad (z) \leftarrow \text{bin}(x) \\ & \quad \quad \quad (y) \leftarrow \text{bin}(z) \\ & \quad \quad \quad \textbf{return} \quad (y) \\ \text{Leaf}_a & : \quad \text{bin}(x) = \textbf{return} \quad (\text{Cons}_a(x)) \end{aligned}$$

using the simplification rules (**SR_i**) given below.

End of Example 5.4.1

(**SR₁**) When the returned value is the result of the last generator, one replaces these two in-

structions by the last call, e.g.:

$$\begin{array}{l} \mathbf{do} \quad (y) \leftarrow \mathit{bin}(\mathit{Nil}) \\ \quad \mathbf{return} \quad (y) \end{array} \Leftrightarrow \mathbf{do} \quad \mathit{bin}(\mathit{Nil})$$

(SR₂) When the **do** sequence is reduced to a unique item –either a call or a **return** statement– one omits the **do** instruction, e.g.:

$$\begin{array}{l} \mathbf{do} \quad \mathit{bin}(\mathit{Nil}) \\ \mathbf{do} \quad \mathbf{return} \quad (\mathit{Cons}_a(x)) \end{array} \Leftrightarrow \begin{array}{l} \mathit{bin}(\mathit{Nil}) \\ \mathbf{return} \quad (\mathit{Cons}_a(x)) \end{array}$$

(SR₃) When the list of conditions in the **where** clause is empty, the clause is omitted, e.g.:

$$\text{Fork :} \quad \mathit{bin}(x)\langle y \rangle \mathbf{where} \quad [] \Leftrightarrow \text{Fork :} \quad \mathit{bin}(x)\langle y \rangle$$

5.4.2.2 Service calls

Multi-user service calls

$$[\mathit{crowd} \mid \mathit{group}] \mathit{sort}^{[+ \mid * \mid \sim]} [role] (\dots)$$

The expression in the generator is preceded by either of the keywords **crowd** or **group** and the sort is superscripted with the kind of multi-user service call, which is either of the following: + (some), * (many), ~ (All). For example:

```
Evaluate_Submission
submission(article) =
  do reports ← crowd evaluate+[reviewer](article)
  decide(reports)
```

Single user service calls

$$\mathit{sort}^{[? \mid ! \mid \#]} [role] (\dots)$$

For example, the following generators respectively model single user service calls to none or one, exactly one, and to a particular user-id.

```
report ← evaluate?[r :: reviewer]
report ← evaluate![r :: reviewer]
report ← evaluate#[user_id]
```

User Inputs In practice, it will frequently be required of the user to provide additional information needed for the processing of the case when they apply rules at open nodes. The **input** clause serves to highlight those parameters of the rule that are instantiated by the user when he chooses this rule. These parameters can also be used for information that the user wishes to preserve in the artifact but which are not necessarily used elsewhere. This clause enriches the GAG syntax and is placed after the equal-to sign (=) and before the **do** keyword. For instance, one may find the following rule in the grammar specification of the reviewer:

```
Accept :
evaluate(article) =
  input (msg :: String)
  do report ← review(article)
  return (Yes(msg, report))
```

With this rule the reviewer informs the Editor that he accepts to review the paper. The returned value is formed with the Yes constructor, acknowledging acceptance, with two arguments: A message and a link to the report that the reviewer commits himself to subsequently produce. Just as with parametric rules, this specification generates an infinite set of actual rules since there exists an infinite number of potential messages –including the empty one.

User input is treated in a similar way to values synthesized by sibling tasks. In fact, one way of interpreting the input clause is to regard it as a subtask that is manually executed by the user. They therefore constitute atomic operations that do not further develop the artifact tree.

We add the following rule to the simplification rules introduced in Sect. 5.4.2.1:

(**SR**₄) We omit the **do** statement if it merely returns the value(s) introduced by the preceding **input** clause:

$$\text{sort}() = \mathbf{input}(x) \mathbf{do} \mathbf{return}(x) \Leftrightarrow \text{sort}() = \mathbf{input}(x)$$

Remark 5.4.2. In practice however, the parameters provided by the user at runtime will correspond either to a specific role in the system –whose instantiations are finite in number– or some kind of data –a message, a report, a decision, etc.– whose values should be kept in the artifact but has no impact on the subsequent behaviour of the system. Therefore, it will be possible to abstract the parameter values to end up with a finite guarded attribute grammar with the same behaviour.

End of Remark 5.4.2

5.4.2.3 Combinators

In order to tailor the specification language towards the applicative domains, we introduce the notion of combinators. Combinators are macro-instructions that encapsulate into easy-to-use constructs, features that are frequently used and which would otherwise require considerable modelling effort from the users.

The set of such useful macro-instructions depend on the application domain. For instance, in disease surveillance, we identify combinators for the following types of tasks:

1. **Iteration:** Recursive business rules of the form, $s \rightarrow \dots s \dots$, are suited for situations where the input values of the inner sort s are computed from the outputs of sibling tasks. There are however situations where the recursive call just handles the remainder of an attribute on the left-hand side. We introduce an iteration schema given by combinator **iter** for such situations, defined as follows. If s is a sort with inherited and synthesized attributes of respective types a and b , **iter** s stands for a new sort whose inherited and synthesized attributes are lists of elements of type a and b respectively, associated with rules

$$\begin{array}{l}
 \mathbf{iter} \ s :: a^* \rightsquigarrow b^* \\
 \mathbf{when} \ s :: a \rightsquigarrow b
 \end{array}
 \equiv
 \begin{array}{l}
 \mathbf{iter} \ s \ (\underline{Cons} \ (\mathbf{head}, \mathbf{tail})) = \mathbf{do} \ \mathbf{h}' \leftarrow s \ (\mathbf{head}) \\
 \mathbf{t}' \leftarrow \mathbf{iter} \ s \ (\mathbf{tail}) \\
 \mathbf{return} \ (\underline{Cons} \ (\mathbf{h}', \mathbf{t}')) \\
 \mathbf{iter} \ s \ (\mathbf{Nil}) = \mathbf{return} \ (\mathbf{Nil})
 \end{array}$$

This is only syntactic sugar: a GAG uses a set of sorts with no *a priori* structure, but one can equip the set of sorts with a set of combinators together with a type system to constraint their usage –for instance **iter** $s :: a^* \rightsquigarrow b^*$ when $s :: a \rightsquigarrow b$ – and with rules which conform to this type system.

2. **Dialog:** It is common to find users engage in a "live" communication during process execution. Such cases occur when one user has to solicit the same service from another user several times with different inputs. For example, during contact tracing in an Ebola epidemic situation, the contacts are progressively identified, and a field worker assigned to trace him/her. Figure 5.2 illustrates contact tracing with two tasks, **Identify contact** and **Trace contact**, the former supplying contact information to the latter which after tracing returns the results to the former.

Example 3.4.2, specifies such communication using lazy lists. We define the **dialog** combinator for two recurring tasks that communicate through lazy lists using the **iter** com-

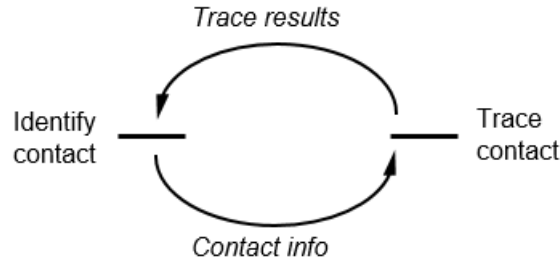


Figure 5.2 – Dialog between two tasks during Contact Tracing

binator as follows:

$$\begin{array}{l}
 \mathbf{dialog} \langle s, s' \rangle :: () \rightsquigarrow () \\
 \mathbf{when} \quad s :: a \rightsquigarrow b \\
 \quad \quad s' :: b \rightsquigarrow a
 \end{array}
 \equiv
 \begin{array}{l}
 \langle s, s' \rangle = \mathbf{input} (x :: a) \\
 \mathbf{do} \quad ys \leftarrow \mathbf{iter} s (\underline{Cons} (x, xs)) \\
 \quad \quad xs \leftarrow \mathbf{iter} s' (ys) \\
 \mathbf{return} ()
 \end{array}$$

The combinator $\mathbf{dialog} \langle s, s' \rangle :: () \rightsquigarrow ()$ uses no information from and returns no information to its surrounding environment. We can improve on this example by replacing this combinator by a (potentially infinite) set of combinators using functions as extra parameters:

$$\begin{array}{l}
 \mathbf{dialog} \langle s, s' \rangle \mathit{in} \mathit{out} :: c \rightsquigarrow d \\
 \mathbf{when} \quad s :: a \rightsquigarrow b \\
 \quad \quad s' :: b \rightsquigarrow a \\
 \quad \quad \mathit{in} :: c \rightarrow a \\
 \quad \quad \mathit{out} :: a^* \times b^* \rightarrow d
 \end{array}
 \equiv
 \begin{array}{l}
 \mathbf{dialog} \langle s, s' \rangle \mathit{in} \mathit{out} = \\
 \mathbf{do} \quad ys \leftarrow \mathbf{iter} s (\underline{Cons} (\mathit{in} x, xs)) \\
 \quad \quad xs \leftarrow \mathbf{iter} s' (ys) \\
 \mathbf{return} (\mathit{out} (xs, ys))
 \end{array}$$

This combinator has two parameters given by variables in and out and thus it actually generate a potentially infinite set of rules according to the actual functions used to instantiate the parameters. Still, if we assume that all such parametric combinators are totally instantiated in a global GAG specification then we guarantee that we end up with a finite specification.

For instance, we may define the derived combinator:

$$\begin{array}{l}
 \mathbf{weave} [s, s'] = \mathbf{dialog} \langle s, s' \rangle \mathit{id} \mathit{zip} :: a \rightsquigarrow (a \times b)^* \\
 \mathbf{when} \quad s :: a \rightsquigarrow b \\
 \quad \quad s' :: b \rightsquigarrow a
 \end{array}$$

where $\mathit{id} :: a \rightarrow a$ is the identity function and $\mathit{zip} :: a^* \times b^* \rightarrow (a \times b)^*$ is the function

given by:

$$\begin{aligned} \text{zip Nil } ys &= \text{Nil} \\ \text{zip } xs \text{ Nil} &= \text{Nil} \\ \text{zip Cons}(x, xs) \text{ Cons}(y, ys) &= \text{Cons}((x, y), \text{zip } xs \text{ } ys) \end{aligned}$$

This combinator can equivalently be specified as

$$\begin{array}{ll} \mathbf{weave} [s, s'] :: a \rightsquigarrow (a \times b)^* & \mathbf{weave} [s, s'] = \mathbf{do} \text{ } \mathbf{ys} \leftarrow \mathbf{iter} \ s \ (\underline{Cons} \ (x, xs)) \\ \mathbf{when} \ s :: a \rightsquigarrow b & \equiv \ \mathbf{xs} \leftarrow \mathbf{iter} \ s' \ (\mathbf{ys}) \\ \ \ \ \ \ s' :: b \rightsquigarrow a & \ \ \ \ \mathbf{return} \ (\mathbf{f} \ \text{zip} \ \mathbf{xs} \ \mathbf{ys}) \end{array}$$

Notice the use of functions *id* and *zip* in semantic rules (see Section 5.4.1.2).

3. **Checklist:** Check-lists are used to send more than one task to a user in a single service call. For instance, when an epidemiologist needs to send a list counter measures to be implemented by a field worker during outbreak management. We define a **checklist** combinator specified as follows:

$$\begin{array}{ll} \mathbf{checklist} [s_1, \dots, s_n] :: (a_1, \dots, a_n) \rightsquigarrow (b_1, \dots, b_n) & \equiv \ \mathbf{checklist} [s_1, \dots, s_n] = \\ \mathbf{when} \ s_i :: a_i \rightsquigarrow b_i & \ \ \ \ \mathbf{input} \ (uid, x_1, \dots, x_n) \\ & \ \ \ \ \mathbf{do} \ y_1 \leftarrow s_1^\# [uid] (x_1) \\ & \ \ \ \ \dots \\ & \ \ \ \ y_n \leftarrow s_n^\# [uid] (x_n) \\ & \ \ \ \ \mathbf{return} \ (y_1, \dots, y_n) \end{array}$$

The caller chooses the user *uid* to whom the list of tasks will be sent and inputs all required data x_1, \dots, x_n . The user should belong to the roles associated with the services s_1, \dots, s_n .

5.4.3 An Example - Flu Outbreak Surveillance

In this section, we illustrate the notations presented in the previous sections on a collaborative case management system. We model the second scenario of Section 1.2 in which three sets of actors with distinct roles (Epidemiologist, Physician, Biologist) actively participate in surveillance Flu surveillance. We model the grammars for the services associated with each of the roles.

Role of a Physician The physician receives patients, clinically examines them to obtain the signs and symptoms which he then checks against the Flu declaration criteria and eventually registers them into the weekly Flu register. At the end of the surveillance week, all registered suspect cases are declared to the epidemiologist.

R1 – Patient Visit :

```
visit (name, gender, dob (day, month, year)) =
  do (symps) ← clinicalAssessment (patient (name, gender, year))
     (rep) ← checkFluCriteria (symps, f age year)
     (careRep) ← initialCare (patient (name, gender, year), symps)
  return (report (symps, rep))
```

Notice the use of simplification rule **SI**₃ to omit the where clause since no conditions are posed on the input variables. The second generator of the **do** statement computes the age of the patient from his/her year of birth – **f age year** – before passing as an attribute of the checkFluCriteria task.

R2 – Clinical Assessment :

```
clinicalAssessment (patient) =
  do input (symps, temperature)
  return (symptoms (symps, temperature))
```

Note that we use here the simplification rule **SR**₄ in **R2** meaning that the input value –the symptoms– is returned as a result of the clinical assessment.

R3 – Check and Declare :

```
checkFluCriteria (name, symptoms (symps, temp), age)
  where [(age < 5 ∧ "cough" ∈ symps)
        ∨ (age ≥ 5 ∧ ("cough", "fever") ⊆ symps) ∧ temp ≥ 38)] =
  do input (site_id)
     ack ← caseDeclaration![epidemiologist] (site_id, name, age,
                                             symptoms (symps, temp))
  return (ack)
```

The evaluation of the guard of rule **R3** succeeds if the patient is aged less than 5 years and has cough as a symptom, or, if the patient is 5 years or older and has cough and fever as symptoms with a body temperature greater than 38°. To effectively declare the suspect case, this rule invokes the caseDeclaration service of the epidemiologist with the suspect case information. Rule **R4** is used when the declaration criteria check fails and no case is declared.

R4 – Do not Declare : checkFluCriteria(name, symps, age) = **return** (-)

R5 – Clinical Examination cont. :

```
initialCare(patient, symps) =
  do input (samples, report)
    labRes ← laboratoryAnalysis2[biologist] (patient, samples)
    return (careRep (report, labRes))
```

The physician continues examining the patient, he extracts saliva samples and sends a laboratory analysis request to a biologist. The returned information - careRep is stored in his workspace and might come in handy during outbreak investigation.

Role of the Biologist The biologist receives the samples, verifies their conditioning and runs the requested analyses. The results are returned for used in confirming or revoking the outbreak alarm. For simplicity, we suppose and only model the case where the sample is well conditioned in which case the corresponding grammar is reduced to rule

R6 – Lab Analyses : **laboratoryAnalysis**(patient, samples) = **input** (labResult)

Role of the Epidemiologist The epidemiologist offers two services: caseDeclaration which is used by physicians to declare suspect cases of Flu, and alarmInvestigation which periodically analysis the reported to detect and investigate outbreak alarms.

1. Case Declaration

The epidemiologist verifies every reported data for errors and when the verification succeeds, the data is automatically stored in a surveillance database – using the function call: "f register name age symps temp" in rule **R9**. When on the other hand the verification fails, the data is not stored, and the errors are sent back to the declaring physician for correction and re-declaration.

R7 – Case Declaration :

```
caseDeclaration (site_id, name, age, symps) =
  do verif ← verifyData (age, symps)
    ack ← storeCaseData(verif, site_id, name, age, symps)
    return (ack (verif, ack))
```

R8 – VerifyData : verifyData (age, symps) = **input** (verif (decision, errors))

R9 – Store case data :

```
storeCaseData(verif (True, -), site_id, name, age, symps) =
  f register name age symps temp
```

Here we have replaced variable *errors* by a dummy variable (the wildcard) to stress that this variable is not used in the rule.

R10 – Report errors in declared data :

```
storeCaseData(verif (False, errors), site_id, name, age, symps) =
  return (errors, name, age, symps)
```

2. Alarm Investigation

Periodically, automated analysis are run against the declared data to check for unexpected hikes above the expected threshold. Such analysis used specialised algorithms which are either programmed in external systems accessible to the process model or as semantic functions in the GAG specification.

R11 – Periodic Analysis :

```
automatedAnalysis () = do input (params)
  recur[2d] autoAnalysis (params)
```

We use the **recur** clause with parameter 2d indicating that the autoAnalysis task should be executed every two days.

R12 – Outbreak Alarm :

```
autoAnalysis (params) = do alarm ← f runDetectionAlgo params
  investigateAlarm (alarm)
```

Each time the automated analysis is run, the epidemiologist's investigateAlarm task is invoked with the alarm object generated by the runDetectionAlgo function.

We suppose that when an alarm is raised, the corresponding alarm object contains amongst other information, the following data: the concerned population, the date of onset of the alarm, the expected number of cases, and the concerned list of sites that declared the alarm data. If however no alarm is raised, the function runDetectionAlgo returns a noAlarm object.

R13 – Investigate Alarm :

```
investigateAlarm (alarm (pop, onset, sites) =
  do alert ← diagnosticAnalysis (pop, onset, sites)
     ack  ← notifyAuthorities (alarm (pop, onset, sites), alert)
     ()   ← initiateRiposte (alarm (pop, onset, sites), alert)
  return (ack (alert, ack))
```

R14 – No Alarm Raised : investigateAlarm (noAlarm) = return ()

R15 – Diagnostic Analysis for Alarm Investigation :

```
diagnosticAnalysis (alarm (pop, onset, sites)) =
  do sitesData ← iter obtainDiagnosticData (sites)
     ack       ← f storeData sitesData
     alert     ← f runAlertDetectionAlso ack
  return (ack (alert))
```

We use the **iter** combinator in **R15** to model an iterative execution on a list of values, **sites**, of the task **obtainDiagnosticData** which sends two tasks **consultationReport** and **laboratoryResults** using a **checklist** to each of the sites in the alarm data – **R16**. We suppose that the function **runAlertDetectionAlso** return an *alert* object when an outbreak alert is detected, otherwise it returns a *noAlert* object.

R16 – Obtain Diagnostic Data :

```
obtainDiagnosticData (site) = do checklist consultationReport (onset)<rep>
                               laboratoryResults (onset)<res>
```

The epidemiologist therefore iteratively indicates a physician in each site to whom the checklist with two service calls will be sent.

R17 – NotifyAuthorities :

```
notifyAuthorities (alarm (pop, onset, sites), alert) =
  do input (mail_list)
     ack1 ← f sendAlarm mail_list, pop, onset, sites
     ack2 ← sendOutbreakAlert (alert, mail_list)
  return (ack (ack1, ack2))
```

R18 – Do Not Notify : notifyAuthorities (noAlarm, _) = return ()

Chapter 6

An Active-Workspaces Prototype and Example

In this chapter, we present a work in progress and proof of concept prototype of the Active-Workspaces model. The prototype is built in Haskell [100], an advanced purely functional programming language with a powerful and intuitive syntax that closely resembles our declarative formalism for guarded attribute grammars.

We start by describing the general architecture of an active workspace system which we implemented in the prototype, then we present the design of an internal domain specific language (DSL) into Haskell for GAG specification and an execution engine that enforces the GAG/AW operational semantics. We end with a presentation of the user interface and how the actions it provides concur to realise the GAG/AW operational semantics.

6.1 Description and Architecture

An Active-Workspaces system functions in near peer-to-peer fashion with connected and disconnected modes communicating via asynchronous message exchanges. More precisely, the users can continue enacting process models while disconnected from the rest of the system and when they eventually connect, all pending communication with other Active-Workspaces is effected. In disconnected mode, outgoing messages are piled up in the output buffer of the emitter's workspace and forwarded when the workspace connects while in connected mode, messages are automatically channelled to the receiving workspace(s) as soon as they are generated.

In the remainder of this chapter, we consider the term **Active-Workspace** (AW) to refer to the enactment environment of a single service. That is, an AW is defined for every service in a **user's workspace** and identified by the unique couple made up of the user's ID and the

sort of the corresponding service. For instance, if a user with user-id **Bob** offers n services with sorts s_1, s_2, \dots, s_n , then Bob's workspace will contain n Active-Workspaces with unique IDs: $(\text{Bob}, s_1), (\text{Bob}, s_2), \dots, (\text{Bob}, s_n)$.

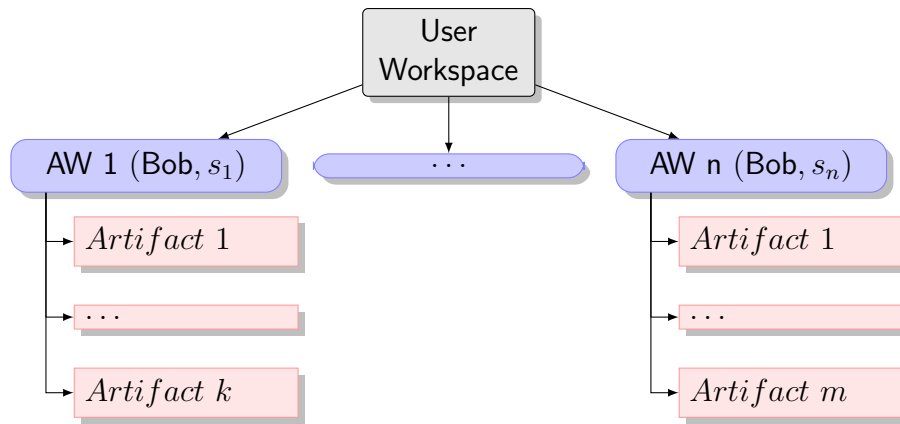


Figure 6.1 – Structural organization of a user's workspace with n active workspaces

A **user's workspace** is therefore a grouping of all the Active-Workspaces for each of the services offered by the user. When a user's workspace is connected, all its contained active workspaces function in connected mode, and vice versa. Figure 6.1 shows the structural organization of Bob's workspace, with active workspaces 1 and n bearing respectively k and m artifacts.

More so, recall from Chapter 4 that AWs and hence their containing user workspaces might be autonomous (defined by autonomous GAGs - see Def. 3.5) or not. In the former case, the GAGs are composed together and will rely on each other to realise certain business goals. Figure 6.2 is a representation of the physical architecture of a composed GAG system. Notice the presence of a server to support asynchronous communication and to store shared data. We discuss the server in detail in the next subsection.

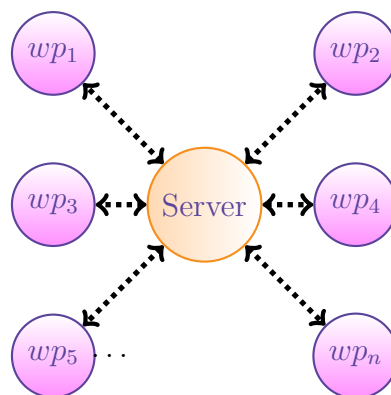


Figure 6.2 – Communicating user workspaces via a server that routes messages and pools shared data

6.1.1 The Active-Workspace Server

Certain non-functional and/or transversal features of the Active-Workspaces system, such as: user management, service management, service assignment/withdrawal to/from users, and message channelling and delivery, need to be centrally provided and accessible to the different user workspaces.

The active-workspaces *Server* (center of Figure 6.2) provides these features. Note that, unlike with classical client/server architectures where the server handles client requests and replies with appropriate responses, the server in the AW system is simply a middleware that facilitates communication between workspaces and stores shared information. We place the features provided by the AW Server into two major groups: Pooling and Routing.

1. **Pooling:** the server stores a pool of users and services as well as the current assignments of services to users. It keeps track of the status (connected/disconnected) of all workspaces. Proceeding this way requires that there existence of an administrator who creates new services or defines existing local sorts in user workspaces as services, adds new users, and modifies assignments of services to users. In this work we consider this centralized functional architecture. However, we can imagine a more decentralized and collaborative approach that employs, data & process mining techniques as well as decision making techniques to identify and adopt new services.

A user can be in one of the following four states:

- (a) **Unenrolled:** the user's account is deactivated or suspended, and the user does not show up in the list of potential service recipients during process enactment.
- (b) **Enrolled:** the user's account is activated or enabled, and the user can receive service calls.
- (c) **Disconnected:** the user's account is enabled but the user's workspace is not connected. Any communication destined for this user will have to be buffered till the user finally connects.
- (d) **Connected:** the user's account is enabled, and the user's workspace is connected. Communication between connected workspaces is Point-to-Point.

Users are Enrolled and UnEnrolled by the administrator. Connecting and Disconnecting a user's workspace on the other hand is done by the user (owner of the workspace).

2. **Routing:** the server serves as a store and router for messages transiting between workspaces. It stores messages destined for currently disconnected users and delivers them when the users connect. This enforces functioning in connected/disconnected modes.

Algorithm 3 presents what happens when a message is to be sent from one workspace to another. If the sender and receiver are connected, communication is point-to-point and

messages are directly exchanged between them. If however, one of them is not connected at the time the message is generated, the message is placed either in the servers buffer if the sender is connected or otherwise in the senders output buffer. The server therefore simple stores messages destined for currently disconnected workspaces.

When a workspace becomes connected, it informs the server of its new status which then forwards all pending messages destined for an Active Workspace in this workspace.

Algorithm 3: Message Exchange between Workspaces

Data: Message m

```

1 if sender is connected then
2   | if receiver is connected then
3     | place  $m$  in receiver's input buffer;
4   | else
5     | place  $m$  in server's buffer;
6   | end
7 else
8   | place  $m$  in senders output buffer;
9 end

```

6.2 A Domain Specific Language for GAG Specification

6.2.1 Domain Specific Languages

A Domain Specific Language (DSL) is a computer language designed for a particular problem domain to improve productivity for designers and communication (and loyalty) of domain experts, as opposed to a General Purpose Language (GPL) that can be used to solve problems across domains [74][40].

In [40] DSLs are divided into three main categories: external DSLs, internal DSLs, and language workbenches.

- An **external DSL** is a language separate from the main language of the application it works with. Usually, an external DSL is a Standalone application with a custom syntax. Examples of external DSLs include regular expressions, SQL, Awk, and XML.
- An **internal DSL** is a particular way of using a general-purpose language. A script in an internal DSL is valid code in its general-purpose language, but only uses a subset of the language's features in a particular style to handle one small aspect of the overall system. Classical examples of this style are Lisp and Ruby on Rails.

- A **language workbench** is a specialized IDE (Integrated Development Environment) for defining and building DSLs. In particular, a language workbench is used not just to determine the structure of a DSL but also as a custom editing environment for people to write DSL scripts. The resulting scripts intimately combine the editing environment and the language.

[74] extends this classification and provides one based on the usage made of the DSL: notation (visual-to-textual or API-to-DSL), AVOPT (analysis, verification, parallelization, and transformation of programs written in a GPL), task automation (eliminate repetitive tasks), product line representation, data structure traversal, interaction (make interactions programmable), and GUI (facilitate building graphical user interfaces).

DSLs are executable in various ways and to various degrees [74]. Executable DSLs capture the behaviour of some underlying model - the semantic model [40]. In our work, we design a DSL for the GAG model with a well-defined execution semantics. Our DSL is an internal DSL implementing an API into Haskell, a GPL with a declarative purely functional syntax and a denotational non-strict semantics, encompassing with it the syntax and semantics of GAGs. It is thus directly executable in Haskell (the host language).

One advantage of using an internal DSL besides obtaining a directly executable language is the ability to make use of the high expressiveness of the host GPL in a coordinated context (the DSL context), to enrich the features of the DSL. The DSL can therefore focus on precisely capturing the structure and behaviour of its semantic model.

We leverage the computational power of Haskell to facilitate data computation and aggregation. Recall from Section 3.1 that we favour a task decomposition methodology that ends with the production of data. Such data can either be manually provided by the user or automatically computed from the inherited attributes and other available contextual data. Being able to use Haskell functions to compute the values of synthesized attributes or to filter and/or aggregate several variables into a single inherited attribute greatly enrich the expressive power of GAG specifications.

Our DSL is intended for designers of collaborative case management systems using GAGs. We intentionally use generic process modelling terminology and vocabulary: tasks, users, data, because it is our intention to develop more precise UI-based DSLs with domain vocabulary, that will be translated into our executable DSL.

In the following sections, we present our design choices as well as the syntax adopted for different parts of a GAG. We subsequently refer to our DSL as the GAG-DSL.

6.2.2 GAG-DSL Generalities

Several data structures and functions are indispensable to understanding the GAG-DSL structure and functioning. We present and expatiate on them in this subsection.

6.2.2.1 The Active Workspace and User Workspace

```
data AW = AW {
    awId      :: AWId           - (userId, service)
    namespace :: IORef Int
    artifacts :: [nodes]
    subscriptions :: IORef (Map Variable [(Variable, AWId)])
    gag       :: GAG
    ... }

```

First, as seen earlier, the `awId`, consisting of the `userId` and the sort of the service, uniquely identifies the AW. Secondly, `namespace` is used during process enactment to: (i) rename formal variables in the specification to local variables in the instance of the applied rule, (ii) count the number of artifacts and their nodes. We use a pointed notation of a list of integers - the Dewey notation -, to identify artifacts, nodes, and variables. A Dewey identifier specifies the exact location of the object in the artifact tree. Thirdly, each AW keeps track of all the `subscriptions` to values of its local variables by variables in distant workspaces. A single local variable can be subscribed to by several distant variables, hence the map `(Map Variable [(Variable, AWId)])` of a variable to a list of variables and their AW identifiers. Finally, the `gag` attribute contains the GAG specification of the service.

```
data UserWorkspace = UW {
    owner      :: User           - a unique String identifier
    services   :: [AW]          - the contained Active Workspaces
    outputQueue :: [Message]
    serviceAssignments :: [(Service, User)] - a Service is identified by its Sort - a String
}

```

The user workspace stores information about the contained AWs, a queue for messages generated while the user workspace is disconnected, and the current assignment of services to users. Recall that a message can be of one of the following forms:

- Service Call: to one or several users
- Subscription: indicate to a distant user that you are subscribing to the value of its variable.
- Sending a value: respond to value subscriptions by sending a value to all subscribers.
- Case Transfer: transfer an artifact from one AW to another.
- Update Subscription: after case transfer, subscriptions are updated so that values are subsequently sent to the appropriate location.

6.2.2.2 The Data

The GAG-DSL supports four types of basic data types: `String`, `Boolean`, `Integer`, and `Double`. This is captured in the following Haskell data type definition:

```
data AWData
  = StringV (Maybe String)
  | BoolV (Maybe Bool)
  | IntV (Maybe Int)
  | DoubleV (Maybe Double)
```

Notice the use of the `Maybe` constructor to allow for variables that contain no values but the promise of a value of the particular type. This is particular useful for the GAG operational semantics in which artifacts can be continuously enacted with holes corresponding to currently unavailable data values, which are filled up when the values become available, making it possible to evaluate expressions, guards, and hence filter applicable rules at open nodes.

```
type Info = [(String, [(AWData, Maybe AWId)])]
```

Information is transported as a list of Key/Value pairs as defined in the `Info` type. We assign a key (`String`) to a vector of values, each coupled with the `awId` of its source AW (`[(AWData, Maybe AWId)]`). This allows for values from multi-user service calls where several AWs each provide a value for a single variable.

6.2.2.3 Inherited and Synthesized attributes

We define two data structures; one (`ValueAW`) that carries data values in tasks and the other (`Pattern`) used in business rules to formally specify patterns on which the data values will be matched to filter applicable rules at open nodes. These data structures implement terms over a ranked alphabet which we use as a record-syntax for inherited and synthesized attributes.

```

data ValueAW
  = Constr String Info [ValueAW]  -- Complex data value with several parts (Records)
  | Var Variable                   -- Intentional data to be provided later on.
  | Val AWDData                   -- Actual data value

```

Inherited Attributes of tasks are constructed using `ValueAW` which model intentional hierarchical data. Note that we Haskell’s algebraic data type to ensure that a data value can be either of the three. That is, a data object is either complex record (`Constr String Info [ValueAW]`), a variable (`Variable`), or an actual value (`AWData`) of any of the four basic types.

Complex objects built with the keyword `Constr` have three parts; the `String` denotes the value constructor, the `Info` which denotes a list of basic data values carried by this record, and `[ValueAW]` which denotes other complex parts of the same data object. For example,

```

Constr "patient"
  [("Id", (StringV "101", _)), ("age", (IntV 37, _))]
  [Var symptoms, Var history]

```

denotes an object of type `ValueAW` that contains information about a patient. This object’s key is the character string `"patient"` and it carries the patients ID and age as basic information. It also contains information about the patient’s symptoms and medical history, which are currently unknown and represented respectively by the variables `symptom` and `history`. We use the underscore wild card (`_`) in the place of the AWIDs since these not specified by the user but are filled up automatically.

```

data Pattern
  = ConstrP String Info [ValueAW]  -- Complex data value with several parts (Records)
  | VarP Variable                   -- Intentional data to be provided later on.

```

Patterns are analogous to objects of type `ValueAW` but carry no actual data values. They are used to match against inherited attributes from tasks. Variables in patterns are pairwise distinct and may be formal or local when the rule is not yet applied at an open node. Just before a rule is applied, all its (formal) variables are renamed to new local variables.

6.2.2.4 Guards

Implicitly matching an object of type `Pattern` to an object of type `ValueAW` at runtime filters applicable rules and initializes variables in patterns. Recall however from Section 5.4.1.1 that

there is need to express more complex filtering conditions and to be able at design time to explicitly initialize certain variables in patterns with values or an aggregation of values extracted from inherited attributes. **Guards** are used to arrive at this result: they implement the concept behind the **where** clause introduced in Section 5.4.1.1 which provides a mechanism through which information extracted during pattern matching can be used for further filtering using comparison operations, and in pure Haskell expressions to initialize input positions of the business rule.

We implement guards as a Haskell (State) Monad¹ and hence write them using the do-notation². The implicit state bears an object of type **Info** corresponding to the substitution produced when pattern matching terminates successfully. This object is omnipresent throughout the body of the do-notation for the guard.

```
newtype Guard a = Guard {runGuard :: Info -> Maybe a}
```

Example 6.2.1. (a simple guard)

Suppose we have a guard with the following *where* clause

```
where (age < 45 & id == "101").
```

It can be expressed as a **Guard** using Haskell's monadic do-notation as follows:

```
do age <- readIntW (<= 45) "age"
   id  <- readStr "id"
   -   <- lg (id == "101")
   return (age, id)
```

This guard does the following:

- **readIntW** checks that the **age** attribute is an Integer at most equal to 45. If the check succeeds, the value is extracted and placed in the variable **pAge**.
- **readStr** reads the value of the **patId** attribute and places it in the variable **pId**
- using comparison operators of the host language Haskell, we can create complex Boolean expressions on the values extracted from the lines above. The results of such expressions, Boolean values, need to be lifted³ back into the **Guard** monad,

1. **Haskell Monad:** A monad is a way to structure computations in terms of values and sequences of computations using those values. Monads allow the programmer to build up computations using sequential building blocks, which can themselves be sequences of computations. The monad determines how combined computations form a new computation and frees the programmer from having to code the combination manually each time it is required.

2. **Do-Notation:** Do notation is an expressive shorthand for building up monadic computations. The do notation allows the latter to be written using a pseudo-imperative style with named variables.

3. **Lift:** Lifting is a concept that allows the transformation of a function to a corresponding function in another (more general) setting

hence the use of the function `lg` (List into Guard). In this example, the guard checks that the value contained in `pId` is equal to the string literal "101".

- `return (age, pId)` renders the variables usable at input positions of the rule being specified.

End of Example 6.2.1

One particularity of the monadic `do`-notation is its sequential execution. It is in reality syntactic sugar for an action chaining operation in which the result of an action is automatically passed downstream to the next action in the chain. The `do` notation enriches Haskell's pure chaining operation by assigning the result of each action to a variable and making it available for use in several subsequent actions. This means that we can write guards which start by extracting particular values, write verify complex Boolean expressions on the values, and lift the result back into the Guard monad.

Each line of a guard returns either the value corresponding to the key parameter, after eventually successfully verifying the condition, or the value 'Nothing' if it fails. A guard fails either when the key is not found in the input data or when the value corresponding to the key does not verify the condition. The overall evaluation of a guard is a pair-wise conjunction of the lines of the `do`-block. If any of the lines return 'Nothing', the entire guard fails and returns 'Nothing' and the rule is retrieved from the list of applicable rules at an open node.

6.2.2.5 Constructs for modelling Collaboration

The following data structure (`Provider`) defines constructs used to specify the different forms of collaboration in business rules.

```
data Provider
  = Many           - - Task to be sent to zero or several users specified at
                    runtime
  | Some            - - Task to be sent to at least one user specified at runtime
  | All             - - Task to be sent to all users offering the service.
  | One             - - Task to be sent to a single user chosen at runtime
  | Single {usr :: Usr} - - Task sent to a single user specified at design time
  | Several {usr :: [Usr]} - - Task sent to several users specified at design time
```

6.2.2.6 Persisting data with JSON text files

To ease communication, storage and retrieval of data about AWs, business rules, enacted artifacts, messages, users, and services, we opt to use the standard JSON text format.

6.2.3 The GAG-DSL Syntax

The GAG-DSL defines a set of Haskell functions to specify the different parts of a business rule; the Guards, the left-hand side, the right-hand side, the inherited attributes and the synthesized attributes. We adopt the following naming conventions for the GAG-DSL syntax elements in the rest of the document: all syntax elements are written in `fixed-width` or `monospace` font. Language keywords are written underlined. Sorts, value constructors, and keys are written enclosed in double inverted commas ("`\"`"), while variables with corresponding types in GAG-DSL are written in *italics*. Finally lists are enclosed within square brackets (`[]`), and optional parameters are enclosed in curly brackets (`{ }`).

6.2.3.1 Guards and Predicate Logic Expressions

Guards implement the content of the `where` clause introduced in Section 5.4.1.1. The general syntax for guards is as follows:

```
do  v_1    <-  expr_1
    { v_2    <-  expr_2
      ...
    v_n     <-  expr_n }
  { return  (v_1 {, v_2, ..., v_n}) }
```

A guard starts with the keyword `do`, followed by at least one assignment expression of the form; `v_i <- expr_i`, and ends with a `return` statement which has as arguments a subset of the variables `v_i`. The `return` statement can be omitted when the guard contains only a single assignment expression.

In the following enumeration, we present the syntax for the various expressions that can be substituted for each of `expr_i` depending on the nature of the data and on whether or not the expression includes a condition.

1. Guard expressions for single value keys

Guards on local values or values that originate from single-user service calls.

(a) Unconditional value extraction

```
Syntax:  read[TYPE] "key"
```

Replace the `[TYPE]` with the corresponding type of the data value to be extracted. It can be one of the following: `Int` for integers, `Bool` for Boolean values, `Str` for

character strings, and `Double` for floating point numbers. The `key` is a character string that uniquely identifies the value to be extracted.

Example 6.2.2. *unconditional single value extraction*

```
readInt "age"
readStr "name"
readDouble "height"
readBool "smoker"
```

(b) **Conditional value extraction**

Syntax: `read[TYPE]W condition "key"`

Same as with unconditional value extraction but with an additional parameter: `condition`. The condition can be any Haskell function that takes as parameter a value of type `TYPE` and returns a Boolean value.

Example 6.2.3. *conditional single value extraction*

```
readIntW (<= 45) "age"
readStrW (== "John Doe") "name"
readBoolW (== TRUE) "smoker"
```

Also, to easy working with character Strings, in addition to verifying their equality, we add a `contains` operator, which can be used to check that a character string contains `NONE`, `ANY`, or `ALL` of the substrings in a list.

Example 6.2.4. *Guard with the contains clause*

```
contains ALL symptoms ["cough", "fever", "nausea"]
contains ANY symptoms ["fever", "feverish", "febrile"]
```

The argument `'symptom'` is a variable which must have been extracted earlier in the guard. The following example using this operator in a guard.

```
do symptoms ← readStr "symptoms"
  -         ← contains ALL symptoms ["cough", "fever"]
  return (symptoms)
```

This guard first extracts the symptoms into a variable `symptom`, checks that its value contains `ALL` of the substrings in the list, then returns the symptoms if the check is successful.

End of Example 6.2.4

2. Guard expressions for multi-valued keys

Guards for values that originate from multi-user group based and/or crowd-based service calls.

(a) Unconditional value extraction

Syntax: `readA[TYPE] "key"`

Recall that a **key** for such values identifies not a single value but an array of values of the same type. Just as with extracting single-valued keys, the [TYPE] in the syntax is replaced with the corresponding type for the data value to be extracted.

Example 6.2.5. *unconditional value extraction*

```
readAInt  "labResult"
readABool "userDecision"
```

(b) Conditional value extraction

Syntax: `readA[TYPE]W quantifier condition "key"`

Recall from Section 5.3.1 that we do not directly pose conditions on the individual data values but are interested in aggregating the data in these arrays and posing constraints such as: "there exist $r \in R$ such that $y[r]$ satisfies \mathcal{C} " – \mathcal{C} being some condition on values of the vector y – or "for all $r \in R$, $y[r]$ satisfy \mathcal{C} ", or even "there exist at least 3 individuals $r \in R$ such that $y[r]$ [satisfy \mathcal{C}]", or if vector y holds Boolean values, "at least 50% of $r \in R$ verify $y[r]$ ". We use three types of quantifiers:

- i. `ForAll` or `every`: all the values in the array must satisfy the condition.
- ii. `(Exists n)` or `(exists n)`: at least n values in the array satisfy the condition.
- iii. `(Proportion d)` or `(prop d)`: at least $d\%$ of the values in the array satisfy the condition.

The `condition` is same as with single value conditional data extraction. It can be any Haskell function that takes as parameter a value of type `TYPE` and returns a Boolean value. One other condition is the `lcontain` (list contain) operator, analogous to the `contain` operator with the difference that it operates on a list of String values. It checks that each of the elements in the list contain all/any/none of a set of substrings.

Example 6.2.6. *conditional value extraction*

```

readAIntW every (< 1990) "yearOfBirth"
readAStrW (exists 5) (== "Male") "gender"
readABoolW (prop 50.0) (== TRUE) "confirmation"
readAStrW (prop 50.0) (lcontain ANY ["Male", "Masculine"]) "gender"

```

Now we describe the syntax for the `when-nb` operator introduced in Section 5.3.1 for crowd-based service calls.

```
Syntax:  guard  'when'  (nb "key" decile)
```

Such expressions delay the evaluation of the `guard` until `decile %` of the values for the `key` have been received. The left parameter `guard` is any multi-value guard expression without its `key` parameter. This is made possible by Haskell currying which allows to have partially instantiated functions, the so called higher order functions.

Example 6.2.7. *guard with the "when nb" operator*

```
readAIntW every (< 1990) 'when' (nb "yearOfBirth" 60)
```

This expression waits for 60% of the responses for the key "yearOfBirth" to be received, then tests if all of them are strictly less than 1990.

3. Building complex Boolean expressions

Variables extracted in guard expressions are as much variables of the host language Haskell as they are variables of the GAG-DSL. They can therefore be used to build any form of Boolean expression that is valid in Haskell. We use the function `lg` (lift guard) to lift the results of such expressions into the Guard monad.

```
Syntax:  lg  (boolExpression)
```

6.2.3.2 The Left-Hand Side - Business Rule Profile

The left-hand side (lhs) or the profile of a business rule is made up of the sort for the rule, its inherited attributes, its synthesized attributes.

```
Syntax:  lhs  "sort"  [inh1, inh2, ...]  [syn1, syn2, ...]
```

Each of `inhi` and `syni` are respectively of type `Pattern` and `ValueAW`. We will see in subsequent sections how these values are built.

When a rule has inherited attributes with no synthesized attributes, the case of information rules that produce no output, we use the following syntax.

```
Syntax:  lhs_  "sort"  [inh1, inh2, ...]
```

When a rule has synthesized attributes and no inherited attributes, we use the following syntax:

```
Syntax:  lhs'  "sort"  [syn1, syn2, ...]
```

When the rule has neither inherited nor synthesized attributes, the case of terminal rules that are used to close open nodes and terminate process enactment, we use the following syntax:

```
Syntax:  lhs_  "sort"
```

6.2.3.3 The Right-Hand Side

The right-hand side (rhs) of a business rule is made up essentially of the list of subtasks for the task. If the concerned rule contains predicate logic based guards, (rules specified by the `rule` or `tRule` keywords below), the following syntax is used:

```
Syntax:   $\lambda(v_1, v_2, \dots) \rightarrow$  rhs  [ task1, task2, ... ]
```

The v_i are variables extracted in the guard expressions. Passing them this way makes them available for use to initialise input positions of the tasks *task_j*.

If however the business rule contains no predicate based guards, (rules specified by the `rule_` or `tRule_` keywords below), the following syntax is used:

```
Syntax:  rhs  [ task1, task2, ... ]
```

6.2.3.4 Tasks: Semantic Rules in the RHS

A task bears not only information about its input and output attributes but also about whether or not it is a local task or a service call. We distinguish below the two groups of tasks:

1. Local Tasks:

The keyword `self` is used to specify tasks that are executed locally in the AW in which their containing GAG is found.

```
Syntax:  self  "sort"  [inp1, inp2, ...]  [out1, out2, ...]
```

Each of *inp_i* and *out_i* are respectively of type `ValueAW` and `Pattern`. We will see in subsequent sections how these values are built.

When a local task does not return any output, the following syntax is used:

```
Syntax: self_ "sort" [inp1, inp2, ...]
```

When on the other hand a task requires no input, but produces output values (constant tasks which return the same value all the time), the following syntax with the keyword `auto` is used:

```
Syntax: auto [valAW1, valAW2, ...]
```

Notice that in such cases, the designer explicitly constructs a list of `ValueAW` objects eventually using variables extracted during guard evaluation.

Also, to allow for manual work done by the user and results injected back into the artifact, or input values that need to be provided by a user when a rule is applied at an open node, the following `input` macro is used.

```
Syntax: input "valueConstructor" [inpVal1, inpVal2, ...]
```

When a business rule contains the `input` clause in its RHS, these are first executed before the other tasks. The values are assembled into a `ValueAW` object with constructor `valueConstructor` and handled in the way as inherited attributes of the business rule. The `inpVali` objects are built using the `[TYPE]_i` keywords presented in Section 6.2.3.5.

2. Service Calls

(a) Crowd-Based Service Calls

i. User Specified at Design Time

— To One User

```
Syntax: to "userId" "sort" [inp1, inp2, ...] [out1, out2, ...]
```

Same as with local tasks but with an additional parameter, the `userId` of the user to receive the service call. If the service call produces no results, use the following syntax:

```
Syntax: to_ "userId" "sort" [inp1, inp2, ...]
```

— To Several Users

```
Syntax: toSeveral "userId" "sort" [inp1, inp2, ...] [out1, ...]
```

Again, if the service call produces no results, use the following syntax:

```
Syntax: toSeveral_ "userId" "sort" [inp1, inp2, ...]
```

ii. **User Specified at Runtime** The information about the users offering the service of sort `sort` is contained in the user's workspace and is automatically used to display the list of potential users when a business rule with any of the following clauses on its RHS is applied at an open node.

— To One User

```
Syntax: toOne "sort" [inp1, inp2, ...] [out1, out2, ...]
```

If the service call produces no results, use the following syntax:

```
Syntax: toOne_ "sort" [inp1, inp2, ...]
```

— **To None or Several Users (Many)**

```
Syntax: toMany "sort" [inp1, inp2, ...] [out1, out2, ...]
```

If the service call produces no results, use the following syntax:

```
Syntax: toMany_ "sort" [inp1, inp2, ...]
```

— **To At least One User (Some)**

```
Syntax: toSome "sort" [inp1, inp2, ...] [out1, out2, ...]
```

If the service call produces no results, use the following syntax:

```
Syntax: toSome_ "sort" [inp1, inp2, ...]
```

— **To All Users**

```
Syntax: toAll "sort" [inp1, inp2, ...] [out1, out2, ...]
```

If the service call produces no results, use the following syntax:

```
Syntax: toAll_ "sort" [inp1, inp2, ...]
```

(b) **Group-Based Service Calls**

Group based service calls are not programmed in the current version of the prototype since they require to implement decision making strategies which we intend to develop as a perspective of this thesis. However, a probable syntax for group based service calls could be:

```
Syntax: toGroup "sort" {"dec_strategy"} [inp1, inp2, ...] [out1, ...]
```

Where the optional parameter `dec_strategy` is the name of the decision making strategy to use in aggregating results. If it is not provided, then it would have to be chosen by the users in the group. If the service call produces no results, the following syntax could be applicable:

```
Syntax: toGroup_ "sort" {"dec_strategy"} [inp1, inp2, ...]
```


6.2.3.5 Defining Values and Terms

1. Values - ValueAW

Objects of type `ValueAW` are used in synthesized positions of the left-hand side of business rules (syn_i above) and in inherited positions of tasks in the right-hand side (inp_i above). These objects are instantiated terms which actual data values. They are specified using the following syntax:

```
Syntax:
ivalue "valConstructor" [info1, info2, ...] [(ivalue ... | var- ...), ...]
OR
var- "key"
```

An AW value has three parts: the value constructor `valConstructor`, a list of information objects ($info_i$) each bearing data of any of the four basic types, and a list of other AW values denoting other parts of the object. Variables in values are defined using the `var`₋ keyword. Recall that synthesized attributes in the left-hand side are simply variables representing promises or placeholders for values to be provided later on.

When an AW value is reduced to its constructor and a list of $info_i$ values, the following syntax is used:

```
Syntax: ivalue- "valueConstructor" [info1, info2, ...]
```

When an AW value is reduced to its constructor and other parts without any $info_i$ objects, the following syntax is used:

```
Syntax: ivalue' "valueConstructor" [(ivalue ... | var- ...), ...]
```

The following simplified statements are used when an AW value carries a single data value in its info part and has no other parts.

```
Syntax: [TYPE]v "valueConstructor" value
```

Where `TYPE` can be any of `int`, `str`, `double`, or `bool` and `value` is a value of the corresponding `TYPE`.

2. Info values

The String/AWData pairs carried by AW values are specified using the following syntax:

```
Syntax: [TYPE]- "key" value
```

Where `TYPE` can be any of `int`, `str`, `double`, or `bool` and `value` is a value of the corresponding `TYPE`. Examples include: `int- "age" 45`, `str- "gender" "Male"`, `bool- "married" True`, `int- "temperature" 32.5`.

Example 6.2.8. *active workspace value object for a patient*

consider the following ValueAW object for a patient:

```
Constr "patient"
  [("Id", (StringV "101", -)), ("age", (IntV 37, -))]
  [Var "symps", Var "hist"]
```

Its corresponding definition using GAG-DSL syntax is:

```
ivalue "patient" [str_ "Id" "101", int_ "age" 37] [var_ "sympms", var_ "hist"]
```

End of Example 6.2.8

When such objects are used to specify empty values, which have to be defined by the user during rule application, the following syntax and keywords are used instead:

```
Syntax: [TYPE]_i "key"
```

3. Terms - Pattern

Recall that patterns are terms over a ranked alphabet used in inherited positions of the left-hand side business rules (inh_1 above) and synthesized positions of tasks in the right-hand side (out_i above). They are terms or variables written using the following syntax:

```
Syntax:
term "valueConstructor" [(term ... | var ...), ...]
OR
var "key"
```

A **term** needs two parameters; the "valueConstructor" and a list of other terms and/or variables denoting other parts of the data structure. (**var**) on the other hand is used when the inherited attribute does not filter the received value. A term with no other parts, that is reduced to its constructor is written with the following syntax:

```
Syntax: term_ "valueConstructor"
```

6.2.3.6 Specifying a Business Rule

A business is specified by defining its various parts (Guard, LHS, and RHS) using the corresponding syntax above. A business rule is a Haskell function that evaluates a guard on values extracted from inherited attributes, returning either an empty list or the list of subtasks as well as the values or variables that will be synthesized from them. Its general syntax is thus:

```
Syntax: rule "Rule Name" LHS Guard RHS
```

If a rule is only structurally guarded and needs no value extraction and verification (no predicate logic based guards), the following syntax is used:

```
Syntax: rule_ "Rule Name" LHS RHS
```

Terminal rules with a guard and no right-hand sides are defined using:

```
Syntax: tRule "Rule Name" LHS Guard
```

Terminal rules with no guard and no right-hand side are defined using:

```
Syntax: tRule_ "Rule Name" LHS
```

Example 6.2.9. *sample business rule - checkFluCriteria*

Consider the following business rule, F3 in Section 5.4.3:

Check and Declare :

```
checkFluCriteria (name, symptoms (symps, temp), age)
  where [(age < 5  $\wedge$  "cough"  $\in$  symps)
          $\vee$  (age  $\geq$  5  $\wedge$  ("cough", "fever")  $\subseteq$  symps)  $\wedge$  temp  $\geq$  38)] =
  do input (site_id)
      ack  $\leftarrow$  caseDeclaration[epidemiologist] (site_id, name, age,
                                                  symptoms (symps, temp))
      return (ack)
```

Using the GAG-DSL syntax, it is specified as follows:

```
R3 = rule "Check and Declare"
  (lhs "checkFluCriteria"
    [var "name", term "symptoms" [var "symps", var "temp"], var "age"]
    [var_ "ack"])
  (do name <- readStr "name"; age <- readInt "year";
    symps <- readStr "symps" temp <- readInt "temp";
    - <- lg ((age  $\geq$  5 && ["cough", "fever"] in symps)
            || (age < 5 && elem "cough" symps);
    return (name, age, symps, temp))
  ( $\lambda$ (name, age, symps, temp)  $\rightarrow$  rhs [
    input site_id [str_ "site_id"]
    toOne "caseDeclaration"
      [var_ "site_id", strv_ "name" name, intv_ "age" age,
       ivalue_ "symptoms" [str_ "symps" symps, int_ "temp" temp ]
       [var "ack"]])
```

End of Example 6.2.9

6.2.4 The GAG-DSL Operational Semantics

A GAG is a list of business rules specified as Haskell function calls using the syntax presented in the previous section. Each of these functions is supplied a task as input and returns as output either an empty list if the corresponding business rule cannot be used to resolve the task (that is, either the sorts don't match, or they match but guard evaluation fails), or a substitution and eventually the (initialised) list of subtasks on the right-hand side of the business rule (if the sorts match and the guard is successfully evaluated).

```
newtype Rules = Rules {gag_ :: Task -> Info -> [(RuleName, GenericRule)]}
```

A Guarded Attribute Grammar (reduced to a single rule) is a function (`gag_`) which when given a task and some initial contextual data, returns a list of grammar rules that can be applied to resolve that task. Each time an open node is created (for some task), this function is run to initialise the node with the applicable rules at that instant. This list of rules is updated as new data is received. Notice that the returned value is a couple `(RuleName, GenericRule)`. The `GenericRule` contains a substitution which is used to initialise variables in its input positions and hence obtain the actual business rule, if the rule is chosen at an open node.

To achieve this behaviour, we use Haskell's `foldl`⁴ function together with our defined weave operator (`<|>`) to realise pairwise weaving of the business rules of the GAG. The output of this operation is a list of partially instantiated functions which will be evaluated against input tasks during process enactment. The `<|>` operator takes as input two rules, prepares them to subsequently evaluate a task argument, and then chains the results of the evaluation using the list concatenation operator (`++`). It is defined as follows:

```
(<|>) :: Rules -> Rules -> Rules
g1 <|> g2 = Rules (\ task info -> gag_ g1 task info ++ gag_ g2 task info)
```

The two arguments (`g1` and `g2`) are single rule grammars which are woven together to form a larger more complex grammar. We assume that no two rules share a common `RuleName` – for easy distinction at the graphical user interface.

4. `foldl`: Higher order Haskell function that systematically combines elements of some data structure (the list of business rules) using a combining function (`<|>`).

To complete a GAG-DSL specification, the designer has to explicitly weave the specified rules. For instance, suppose we have a GAG-DSL specification for some process with four rules `r1`, `r2`, `r3`, and `r4`, we prepare the rules for evaluation as follows:

```
grammar = foldl (<|>) [r1, r2, r3, r4]
```

6.3 Enactment User Interface and Actions

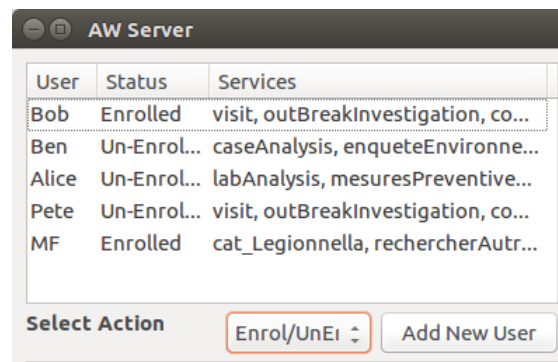




Figure 6.3 – Active Workspaces Server showing the list of users, their status and the list of services they offer.

Figure 6.3 shows a screen-shot of an active workspace server that pools five (5) users. For each of the users, it stores the status (Un-Enrolled, Enrolled, Connected, or Disconnected) and the list of services each offers. Recall that active workspaces are generated only for enrolled users. New users can be added and services assigned to them on-the-fly during execution.

Figure 6.4 shows a screen-shot of the active workspace of the user Bob to whom three services have been assigned - identified by the tabs `AW:visit`, `AW:outbreakInvestigation`, and `AW:confirmAlarm`. On the active tab, `AW:visit`, the left panel contains the list of (two) enacted artifacts for the service. Closed nodes are denoted by  while open nodes are denoted by . Information about the highlighted node is shown on the right panel. In this shot, information about the `caseDeclaration` open node is shown: its internal ID (`[2,2,8]`), its sort `caseDeclaration`, its inherited attributes, its synthesized attributes, and the list of applicable or pending (whose guards have not been completely evaluated) business rules. Notice that the second inherited attribute is a variable, (`Var (Loc "symptoms" 4)`) whose value is still pending, and which is necessary to complete the evaluation of the guards for the rule `Declaresuspectcase`.

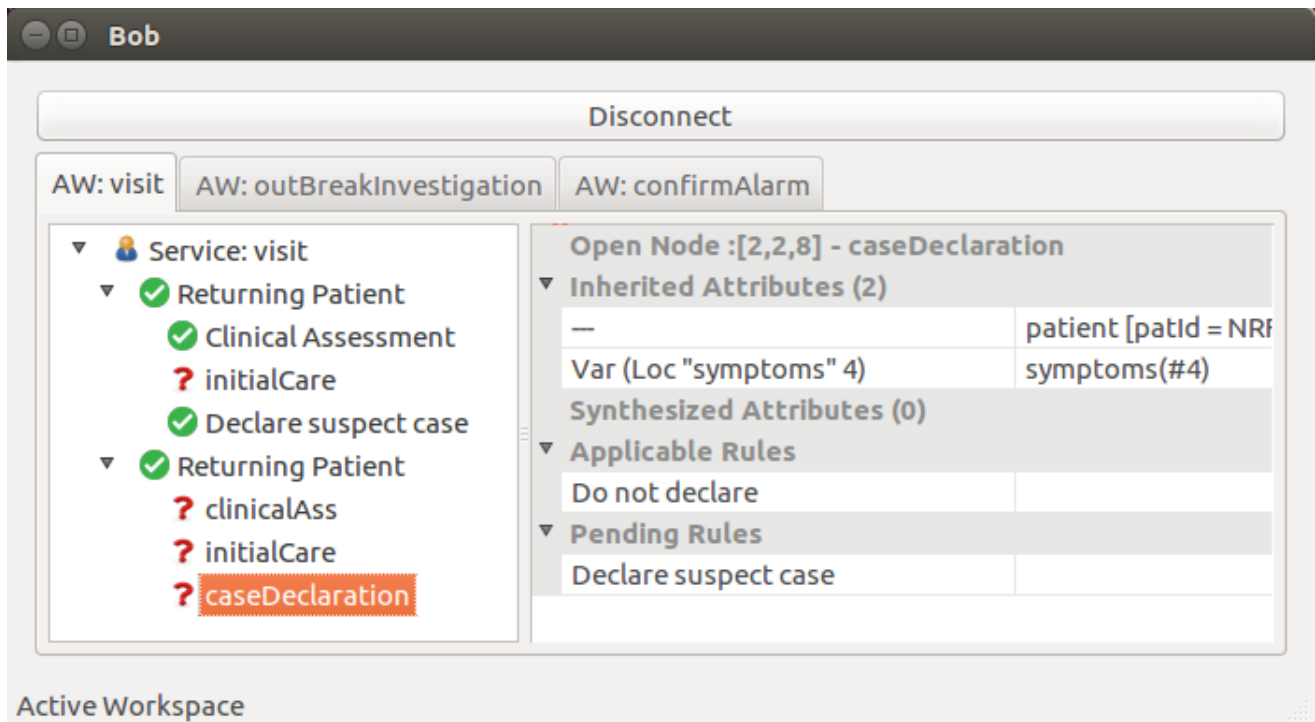


Figure 6.4 – User’s Workspace for User Bob offering three services: visite, outbreakInvestigation, and confirmAlarm

6.4 GAG-DSL Example: Flu Outbreak Management

To illustrate the GAG-DSL syntax and semantics, we rewrite the example described in Section 1.2 and formally specified in Section 5.4.3 of a Flu Outbreak Surveillance system. We then illustrative the GAG semantics on a sample execution with emphasis on the actions available to users. Recall that the modelled example describes the workspaces of three groups of users in a disease surveillance process: physicians who receive, consult, declare, and follow up patients, epidemiologists who monitor the declared data for outbreak alarms, and biologists who provide diagnostic information needed to confirm and/or discard outbreak alarms. In the following sections, we present the GAG specifications for the three groups of users.

1. GAG-DSL Specification for Physicians:

R1 = rule "Patient Visit"

```

(lhs "patientVisit"
  [ var "name", var "gender",
    term "dob" [ var "day", var "month", var "year" ] ]
  [ var "report" ]
)
(do name ← readStr "name"
  year ← readInt "year"
  gender ← readStr "gender"
  return (name, year, gender)
)
( $\lambda$ (name, year, gender) → rhs [
  self "clinicalAssessment"
    [ ivalue "patient" [ str "name" name, str "gender" gender,
      str "year" year ] ]
    [ var "symps" ]
  self "checkFluCriteria" [ strv "name" name, var "symps",
    intv "age" (2017 - year) ]
    [ var "rep" ]
  self "initialCare"
    [ ivalue "patient" [ str "name" name, str "gender" gender,
      str "year" year ],
      var "symps" ]
    [ var "careRep" ]
  auto [ ivalue "report" [ strv "symps" symps, var "rep" ] ]
])

```

Business rule R1 has name "Patient Visit", sort "patientVisit", three inherited attributes "name", "gender" and "dob" and a synthesized attribute "report" on its left-hand side. The only guard on this rule is on the number of inherited attributes (3) and the structure of the `dob` attribute: its sort must be "dob" and it must have three parts respectively for the `day`, `month`, and `year` of birth.

The expressions in the `do`-block simply extract the individual values into variables which are usable to build attribute values in the right-hand side of the rule.

The right-hand side has three local (identified by the use of the keyword `self`) sub-tasks of sorts "clinicalAssessment", "checkFluCriteria", and "initialCare". The `clinicalAssessment` task has one inherited attribute `patient`; a value that carries information (name, gender, & year) and has no other parts, built using the `ivalue` macro, and one synthesized attribute `symps` reduced to a variable. The `checkFluCriteria` task has three inherited attributes: the first and third attributes

(`name` and `age`) are values built using data extracted during guard evaluation, while the second attribute (`symps`) is a variable subscription to the value synthesized by subtask `clinicalAssessment`. Also, notice the use of Haskell's subtraction operation to automatically compute the age from the year of birth (the `f` operator described in Section 5.4.1.2). The third subtask `initialCare` has two inherited attributes: a `patient` value identical to that of the `clinicalAssessment` subtask and a `symps` attribute subscribing to the value synthesized by the same subtask. Finally, notice the use of the `auto` macro to automatically build the synthesized attribute of the rule from the synthesized attributes of its subtasks.

```
R2 = rule_ "Clinical Assessment"
      ( lhs "clinicalAssessment" [var "patient"] [var_ "sympoms"] )
      ( rhs [input "symptoms" [str_i "symps", int_i "temp"]] )
```

Business rule R2 is reduced to a user input. It is assumed that the physician uses complementary tools to the process management system and inputs the obtained results (in this case `symps` and `temp`) into the system. The use of `str_i` and `int_i` macros constrain the provided values to be respectively of types `String` and `Integer`.

This rule is built using the `rule_` macro since it has neither value extraction nor any predicate logic-based guards. It however requires that the task comes with at least one inherited attribute which it places in the `patient` variable.

```
R3 = rule "Check and Declare"
```



```

(lhs "checkFluCriteria"
  [var "name", term "symptoms" [var "symps", var "temp"], var "age"]
  [var_ ack])
(do name ← readStr "name"
  age ← readInt "year"
  symps ← readStr "symps"
  temp ← readInt "temp"
  - ← lg ((age >= 5 && elems ["cough", "fever"] symps)
    || (age < 5 && elem "cough" symps))
  return (name, age, symps, temp))
(λ(age, symps, temp) → rhs [
  input site_id [str_ "site_id"]
  toOne "caseDeclaration"
    [var_ "site_id", strv_ "name" name, intv_ "age" age,
      ivalue_ "symptoms" [str_i "symps" symps, int_ "temp"
        temp ]
    [var "ack"]
  ])

```

Similar in structure to R1, business rule R3 illustrates the use of Haskell's logic operators in conjunction with GAGDSL's custom boolean operators to express predicates on the extracted values. The first four expressions of the do-block extract values into variables and the fifth expression checks that either the extracted `age` value is at least 5, that the strings "cough" and "fever" are contained in the `symps` value, or the `age` value is less than 5 and the symptoms contains at least the "cough" string. Also, notice the one-to-one call to an external service `caseDeclaration` in the right-hand side of R3. When the Flu case declaration criteria is successfully verified, the physician inputs a character string that identifies his surveillance center (`site_id`), which is used alongside other values extracted in the do-block to build the inherited attributes of the task for the service call.

```
R4 = tRule_ "Do Not Declare" (lhs_ "checkFluCriteria")
```

Business rule R4 is a terminal rule that will be automatically invoked during process enactment when the Flu case declaration criteria is not verified. Such rules halt the development of a branch of the artifact tree.

```
R5 = rule "Clinical Examination Cont."
```

```
( lhs      "initialCare" [var "patient", var "symptoms"] [var_ "careRep"] )
  (readStr "symps")
  (λ(symp) → rhs [input "samples" [str_i "samps"],
                 input "report" [str_i "report"],
                 toOne "laboratoryAnalysis" [strv_ "symps" symps,
                                             var_ "samples"]
                                             [var "labRes"],
                 auto [ivalue' "careRep" [var_ "report",
                                             var_ "labRes"]]]
  ])
```

R5 is a complete business rule (that is, it possesses all the parts of a business rule) that uses a simplified guard expression: a do-block with a single expression is reduced to the expression. This rule models work done out of the system by the physician and whose results (`samples` and `report`) are injected back into the process. When this rule is selected, the physician has to provide the two input values before the `laboratoryAnalysis` service call is made. Also, this rule makes the service call in all situations, whereas in surveillance practice, carrying out laboratory analysis is not systematic.

One way to improve the specification is to replace the service call by a local task and define two rules for it; one that makes the service call and the other that simple closes the artifact branch.

We now complete the GAGDSL specification of the physicians' grammar by weaving the defined rules into a list of higher order Haskell functions.

```
physician_gag = foldl (<|>) [R1, R2, R3, R4, R5]
```

This grammar will be available to every user assigned the service with sort `patientVisit`.

2. GAG-DSL Specification for Biologists

```
R6 = rule_ "Lab Analysis"
```

```
(lhs "laboratoryAnalysis" [var "pat", var "samp"] [var_ "labResult"] )
(rhs [input "labResult" [str_i "labResult"]])
```

Reduced to a single rule in this specification, the GAG-DSL specification of the work of a biologist in the context of disease surveillance requires only the final result of the analyses, which we model simply as text.

3. GAG-DSL Specification for Epidemiologists

```

R7 = rule "Case Declaration"

  (lhs "caseDeclaration" [var "sId", var "name", var "age", var "symps"]
    [var_ "ack"])

  (do nm      ← readStr "name"
    age      ← readInt "year"
    symps    ← readStr "symps"
    sId      ← readInt "site_id"
    return (name, age, symps, sId)
  )

  ( $\lambda$ (nm, age, symps, sId) → rhs [
    self "verifyData" [intv "age" age, strv "symps" symps] [var "verif"],
    self "storeCaseData" [var_ "verif", strv "site_id" sId, strv_ "nm" nm,
      intv "age" age, strv "symps" symps]
      [var "ack1"],
    auto [ivalue' "ack" [var_ "verif", var_ "ack1"]]
  ])

```

Business Rule R7 is unguarded and is the only rule with sort `caseDeclaration` hence it is automatically invoked on reception of a service call. The rule verifies the declared data, stores the data if the verification succeeds, and returns an acknowledgement message to the declaring physician, eventually with any errors or incoherences found in the data.

```

R8 = rule_ "Verify Data"

(lhs "verifyData" [var "age", var "symps"] [var_ "verif"] )
(rhs [input "verif" [bool_i "decision", str_i "errors"]])

```

Again, the data is verified in an external tool and the results of the verification (`decision` and `errors`) are entered back into the process. The content of the `errors` attribute will be ignored if the value of the attribute `decision` is `True`.

```

R9 = rule_ "Store Case Data"

(lhs "storeCaseData" [term "verif", var "site_id", var "nm",
  var "age", var
  "symps"]
  [var_ "acknowledgement"])

(readBoolW (== True) "decision")
( $\lambda$ (_) → rhs [auto [strv_ "ack" "OK"]])

```

```

R10 = rule "Store Case Data"

  (lhs "storeCaseData" [term "verif", var "site_id", var "nm",
                        var "age", var "symps"]
      [var "ack"])

  (do dec ← readBoolW (== False) "decision"
      sId ← readStr "site_id"
      nm ← readStr "nm"
      age ← readInt "age"
      symps ← readStr "symps"
      errors ← readStr "errors"
      return(dec, sId, nm, age, symps, errors)
  )

  ( $\lambda$ (dec, sId, nm, age, symps, errors) → rhs [
      auto [ivalue' "ack" [strv "errors" errors, strv "name" nm,
                          intv "age" age, strv "symps" symps]])

R11 = rule_ "Data Analysis"

  (lhs "automatedAnalysis"
  (rhs [input "params" [str_i "params"],
        self_ autoAnalysis [var "params"]]))

R12 = rule "Outbreak Alarm"

  (lhs "autoAnalysis" [var "params"]
  (readStr "params")
  ( $\lambda$ (p) → rhs [self "detectionAlgo" [strv "params" p] [var_"alarm"],
                 self_ "investigateAlarm" [var "alarm"]]))

R13 = rule "Alarm Produced - Start Investigation"

```

```

(lhs "investigateAlarm" [term "alarm" [var "pop", var "onset",var "sites"]]
      [var_ "ack"])
(do  pop    ← readStr "population"
     onset  ← readStr "date_of_onset"
     sites  ← readInt "sites_concerned"
     return(pop, onset, sites)
)
( $\lambda$ (pop, onset, sites) → rhs [
  self "diagnosticAnalysis" [strv "pop" pop, strv "onset" onset,
                             strv "sites" sites] [var
    "alert"]
  self "notifyAuthorities" [ivalue' "alarm" [ var_ "pop" pop,
        var_ "onset" onset, var_ "sites" sites], var_
    "alert"]
    [var "acknowledgement"]
  self_ "initiateRiposte" [ivalue' "alarm" [ var_ "pop" pop,
        var_ "onset" onset, var_ "sites" sites], var_
    "alert"]
  auto [ivalue' "ack" [var_ "alert", var_ "acknowledgement"]]
])

```

```
R14 = tRule_ "No Alarm Raised" (lhs_ "investigateAlarm" [term_ "NoAlarm"])
```

```
R15 = rule "Diagnostic Analysis for Alarm Investigation"
```

```

(lhs "diagnosticAnalysis" [term "alarm" [var "pop", var "onset",var "sites"]]
      [var_ "ack"])
(do  pop    ← readStr "population"
     onset  ← readStr "date_of_onset"
     sites  ← readInt "sites_concerned"
     return(pop, onset, sites)
)
(λ(pop, onset, sites) → rhs [
  self "diagnosticAnalysis" [strv "pop" pop, strv "onset" onset,
                             strv "sites" sites] [var
    "alert"]
  self "notifyAuthorities" [ivalue' "alarm" [ var_ "pop" pop,
    var_ "onset" onset, var_ "sites" sites], var_
    "alert"]
    [var "acknowledgement"]
  self_ "initiateRiposte" [ivalue' "alarm" [ var_ "pop" pop,
    var_ "onset" onset, var_ "sites" sites], var_
    "alert"]
  auto [ivalue' "ack" [var_ "alert", var_ "acknowledgement"]]
])

```

Conclusion

In Part III, we introduced the Active-Workspaces framework with an underlying guarded attribute grammar model. We showed how this new framework supports dynamic process modelling with users and services as the basic building blocks, how it supports flexible process enactment, collaborative work, and timeliness in process management using Timed GAGs.

Then we showed how GAGs can be transformed into executable process models within the AW framework with an extended formalism that supports high level user actions for process control and collaboration (user interactions). We started with an intuitive functional notation inspired from the notation of Haskell's do-blocks, then we designed and implemented a prototype textual (internal) DSL, the GAG-DSL, into Haskell together with a process enactment GUI. The prototype at its current stage only simulates execution on a distributed architecture while on a single computer. In a near future, we intend to effectively implement a distribution component into the prototype and design an easy to use graphical DSL for domain experts with little or no knowledge in computer programming.

To show the applicability of our framework, we ended this part with an example specification of a real-world situation from disease surveillance using the GAG-DSL syntax.

Part IV

Conclusion

Chapter 7

Conclusion, Discussion, and Future Works

Our objective in this thesis was to design a suitable (adaptive case management) model for distributed collaborative systems such as the disease surveillance system. Using examples and literature from disease surveillance systems, we characterized such systems as requiring high levels of flexibility at both design and run-time. The systems are artifact-centric, user-driven, and collaborative, and their modelling should include all of the following aspects: (i) iterative (incremental) design, (ii) collaboration and user interactions, (iii) techniques to leverage uncertainty and exceptions, (iv) decision-making support. We proposed a model that naturally supports iterative modelling, concurrent and flexible process enactment, collaboration and user interactions in a distributed environment. We argued that our model can easily be extended with techniques from cognitive sciences and naturalistic decision-making theories to handle uncertainty and exceptions, and to support (collaborative) decision-making.

The key idea in our declarative model for artifact-centric collaborative systems is to represent the workspace of a stakeholder by a set of (mind)maps associated with the services that the user offers. Each map consists of the set of artifacts created by the invocations of the corresponding service. An artifact records all the information related to the treatment of the related service call. It contains open nodes corresponding to pending tasks that require user's attention. In this manner each user has a global view of the activities in which he is involved, including all relevant information needed for the treatment of the pending tasks.

Using a variant of attribute grammars, called guarded attribute grammars, we automate the flow of information in collaborative activities with business rules that put emphasis on user's decisions. We gave an in-depth description of this model through its syntax, its behaviour, and its main properties. We paid attention to two crucial properties of this model. First, the input-enabled GAG satisfies a monotony property that allows to distribute the model on

an asynchronous architecture. Second, soundness is a property that asserts that any case introduced in the system can reach completion. These properties are both undecidable, but we have defined restrictions on GAGs that guarantee their preservation.

The active-workspaces model encompasses all high-level concepts built on guarded attribute grammars and which are indispensable in dynamic collaborative systems. These concepts include roles, services, task relationships, temporal dependencies, collaboration constructs, and decision support tools. We use the GAG model to show how users are organized into roles, assigned services, and made to collaborate and interact effectively with each other on related time bound tasks.

We transformed the formal notations into a more language-oriented syntax and used it to design a prototype for the model including an internal domain specific language implemented in Haskell, and an enactment user interface that captures all the user actions underlying the Active Workspace model.

Finally, we have demonstrated the expressive power and exemplified the key concepts of active workspaces on a case study for a disease surveillance system – Flu outbreak surveillance –.

To finalize our conclusion, we discuss the key features of our model and draw some future research directions.

7.1 Assessment of the Model

In a nutshell *active workspaces and guarded attribute grammars provide a modular, declarative, user-centric, data-driven, distributed and reconfigurable model of case management*. It favours flexible design and execution of business process since it possesses (to varying degrees) all four forms of *Process Flexibility* proposed in [114].

Concurrency: The lifecycle of a business artifact is implicitly represented by the grammar productions. A production decomposes a task into new subtasks and specifies constraints between their attributes in the form of the so-called semantic rules. The subtasks may then evolve independently as long as the semantic rules are satisfied. The order of execution, which may depend on value that are computed during process execution, need not (and cannot in general) be determined statically. For that reason, this model dynamically allows maximal concurrency. In comparison, models in which the lifecycle of artifacts are represented by finite automata constrain concurrency among tasks in an artificial way.

Modularity: The GAG approach also facilitates a modular description of business processes. For instance, in the scenario described in Section 1.2 and modelled in Chapter ??, when laboratory test requests are sent to the biologist, the physician needs not know about the subprocess through which the specimen will pass before results are finally produced. For instance, the service *laboratoryAnalysis* can –in a more refined specification– be modelled by a large set of rules including specimen purification, and several biological and computational processes. However, following a top-down approach, one simply introduces an attribute in which the results should eventually be synthesized and delegate the actual production of the expected outcome to an additional set of rules. The identification of the different roles involved in the business process can also contribute to enhance modularity. Finally, some techniques borrowed from attribute grammars, like descriptive composition [43, 44], decomposition by aspects [131, 130] or higher-order attribute grammars [122, 106], may also contribute to better modular designs.

Reconfiguration: The workflow can be reconfigured at run time: New business rules (associated with productions of the grammar) can be added to the system without disturbing the current cases. By contrast, run time reconfiguration of workflows modelled by Petri nets (or similar models) is known to be a complex issue [75, 36]. One can also add “macro rules” corresponding to specific compositions of rules. For instance, if the Editor-in-chief wants to handle the evaluation of a paper, he can decide to act as an associate editor and as a referee for this particular submission. However, this means forwarding the corresponding case to himself as an associate editor and then asking himself as a referee if he is willing to write a report. A more direct way to model this decision is to encapsulate these steps in a compound macro production that bypasses the intermediate communications. More generally compound rules can be introduced for handling unusual behaviours that deviates from the nominal workflow.

Logged information: When a case is terminated, the corresponding artifact collects all relevant information of its history. Nodes are labelled by instances of productions that have led to the completion of the case. Hence, they record the decisions and the information associated with these decisions. In the case of the Flu surveillance system, a terminated case contains the names of all the stakeholders (physicians, biologists, and epidemiologists), the data they manipulated, the decisions they made, etc. A terminated case is a tree whose branches reflect causal dependencies among sub-activities used to solve a case, while abstracting from concurrent sub-activities. The artifacts can be collected in a log which may be used for the purpose of process mining [113] either for process discovery (by inferring a GAG from a set of artifacts using common patterns in their tree structure) or for conformance checking (by inspection of the logs produced during simulations of a model or the executions of an actual implementation).

Distribution: Guarded attributed grammars can easily be implemented on a distributed architecture without complex communication mechanisms –like shared memory or FIFO channels. Stakeholders in a business process own open nodes and communicate asynchronously with other stakeholders via messages. Moreover there are no edition conflicts since each part of an artifact is edited by the unique owner of the corresponding node. Moreover, the temporary information stored by the attributes attached to open nodes no longer exist when the case has reached completion. Closing nodes eliminates temporary information without resorting to any complex mechanism of distributed garbage collection.

Interoperability: In Section 5.4.1.2, we describe how the GAG model can be extended with external tools for data storage, basic computations, and even more complex operations such as running data analysis to detect disease outbreaks. We implemented these concepts into the internal DSL in Chapter 6 in the Haskell general purpose language. Such side effects can thus be directly written in the host language. These features generally do not conflict with the GAG specification but rather complement it, in basically two ways. First, they allow to associate real-world activities with a rule, like extracting samples from a patient, sending messages, performing verifications, etc. Second, they may be used to extract information from the current artifacts to build dashboards or to feed some local database that can be used later for a plethora of other purposes.

7.2 Future Work

An immediate milestone is the elucidation of the model by specifying as many (diverse) case studies as possible from a plethora of business domains. These case studies will provide insight needed to fine tune the model and update the prototype environment. One other objective is to move towards a graphical DSL for GAG specifications, necessary to scale the use of the model by (non-technical) domain experts. The following research directions are also considered:

Decision making support and Uncertainty management: Users are faced with uncertain decisions at several levels of process design and enactment. Knowing how to decompose a task into sub-activities (specifying new business rules), choosing which rule to apply at an open node, and deciding on what data to provide as input are all sources of doubt in dynamic processes modelled with GAGs. Information logged in artifacts and/or contained in local databases (execution traces, inherited and synthesized data values, semantic rules, guards, etc.) can be exploited to leverage uncertainty and enhance decision-making. We will explore process mining [116] and discovery [115] techniques coupled with techniques of naturalistic decision-making [66] and cognitive sciences [76], with the objective of augmenting the GAG model with tools

that aid users build and run dynamic processes on-the-fly. Another key aspect worth exploring is collaborative decision-making where each decision is made not by an individual but by a group of individuals. It is useful in group-based service calls.

Development methodology and Collaborative modelling: We need to develop support for the derivation of a GAG specification from a problem description. Object-oriented programming uses, for that purpose, normalized notations and diagrams for specifying the involved classes, use cases, activities and collaborations. A modelling language for GAG should concentrate on the central concepts of the model: The artifacts, task decomposition, user's decisions, user's communication. For the latter one may use concepts of speech act theory [8, 127] for classifying business rules in terms of assertions, orders, requests, commitments, etc. As far as artifacts are concerned, one can observe on the two examples of the paper (Editorial process and Disease surveillance) that we have very few completed artifacts, once the case's specific information contained in the artifacts has been abstracted, one can try to extract the business rules –task decomposition and semantic rules– starting from these archetypal artifacts and answering the following questions: What are the dependencies between data field values? Who produces these values? What information does one need to produce these values? Can one identify the conditions that justify variabilities between similar artifacts? How can business rules local to a user's workspace be made available to others? etc. As a formalism for distributed collaborative systems the GAG model should also come with a complete method for elaborating the procedure going from a problem description, through the implementation, to the deployment on a distributed asynchronous architecture. Just as with Software Processes [101], this method will provide notations which describe how to identify relevant information (roles, data, processes ...) and propose appropriate representation tools to add expressiveness to the textual descriptions of collaborative case management systems.

Improve Process Flexibility: It might become necessary to explicitly indicate that tasks be skipped or hidden during process enactment under certain constraints, or to roll-back (undo) a number of actions taken by the user. This kind of flexibility which is usually embedded into the modelling language's syntax allows a process instance to deviate at runtime from the execution path prescribed by the original process without altering the process definition itself. It is called Flexibility by Deviation [114]. Skip/Hide can be easily achieved with the current GAG model by using guards that enable only terminal rules. However, undo/redo cannot be specified straightforwardly since the enactment of process artifacts can proceed over several (autonomous) user workspaces. Several works have addressed the undo problem in BPM [4, 135, 98]. The GAG model can be augmented to support for atomic transactions (in a distributed setting) with the possibility of rolling back if the transaction does not terminate correctly.

Applicability (More case studies): Our disease surveillance case study was helpful to demonstrate the pertinence of our model and motivate our modelling decisions. More case studies in other dynamic domains are important to identify pertinent (cross-cutting) concepts and refine our specification language. The following three domains are representative case studies in which our model can provide advantages over other modelling approaches: (i) crowd sourcing, where mostly non-expert users collaboratively (consciously or not) resolve tasks. This would require a fine description of user roles and how they can be composed during task resolution to obtain reliable results; (2) reporting systems, where several users collaboratively construct and edit reports; (3) distributed distant learning, a highly decentralised system deployed mostly in degraded environments with limited connectivity. Also, the declarative decomposition of learning activities gives more flexibility in the design and description of learning activities and more freedom to the learner's learning path.

Bibliography

- [1] Serge Abiteboul, Jérôme Baumgarten, Angela Bonifati, Gregory Cobena, Cosmin Cremarencu, Florin Dragan, Ioana Manolescu, Tova Milo, and Nicoleta Preda. Managing distributed workspaces with active XML. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, volume 29, pages 1061–1064. Morgan Kaufmann, San Francisco, 2003.
- [2] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo, and Roger Weber. Active XML: A Data-Centric Perspective on Web Services. In *Web Dynamics: Adapting to Change in Content, Size, Topology, and Use*, pages 275–299. Springer Berlin Heidelberg, 2004.
- [3] Michael Adams, Arthur H. M. ter Hofstede, Wil M. P. van der Aalst, and David Edmond. Dynamic, Extensible and Context-Aware Exception Handling for Workflows. In *On the Move to Meaningful Internet Systems 2007, Vilamoura, Portugal*, pages 95–112. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-76848-7_8.
- [4] Alessandra Agostini, Giorgio De Michelis, and Marco Loregian. Undo in workflow management systems. In *Business Process Management: International Conference, (BPM 2003) Eindhoven, The Netherlands*, pages 321–335. Springer Berlin Heidelberg, 2003.
- [5] Alessandra Agostini, Giorgio De Michelis, Maria Antonietta Grasso, Wolfgang Prinz, and Anja Syri. Contexts, work processes, and workspaces. *Computer Supported Cooperative Work*, 5(2/3):223–250, 1996.
- [6] James Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.
- [7] Pascal Astagneau and Thierry Ancelle. *Surveillance épidémiologique*. Lavoisier, 2011.
- [8] J.L. Austin. *How to Do Things with Words*. Oxford Univ. Press, 1962.
- [9] Kevin Backhouse. A Functional Semantics of Attribute Grammars. In *Tools and Algorithms for the Construction and Analysis of Systems, (TACAS)*, volume 2280 of *Lecture Notes in Computer Science*, pages 142–157. Springer, 2002.
- [10] Badouel, Loïc Hélouët, Georges-Edouard Kouamou, and Christophe Morvan. A Grammatical Approach to Data-centric Case Management in a Distributed Collaborative En-

- vironment. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing, SAC'15*, Salamanca, Spain, pages 1834–1839, 2015.
- [11] Eric Badouel, Loïc Hélouët, Georges-Edouard Kouamou, Christophe Morvan, and Robert Fondze Jr Nsaibirni. Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars. *ACM SIGAPP Applied Computing Review*, 15(3):6–34, 2015.
- [12] Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In *Business Process Management*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–304. Springer Berlin Heidelberg, 2007.
- [13] Berndt Brehmer. Dynamic decision making: Human control of complex systems. *Acta Psychologica*, 81(3):211–241, 1992.
- [14] Emily V Burns, Dave Duggal, Frank Michael Kraft, John T Matthias, Dermot McCauley, Nathaniel Palmer, Jax J. Pucher, Bruce Silver, and Keith D. Swenson. *Taming the Unpredictable: Real World Adaptive Case Management: Case Studies and Practical Guidance*. Future Strategies Inc.;, 1st edition, 2011.
- [15] Diego Calvanese, Giuseppe De Giacomo, Richard Hull, and Jianwen Su. Artifact-Centric Workflow Dominance. In *Service-Oriented Computing: 7th International Joint Conference, ICSOC-ServiceWave 2009, Stockholm, Sweden*, pages 130–143. Springer Berlin Heidelberg, 2009.
- [16] Cristiano Castelfranchi. Modelling Social Action for AI Agents. *Artificial Intelligence*, 103(1):157 – 182, 1998.
- [17] CDC. Centers for Disease Control and Prevention, 2017. <https://www.cdc.gov>.
- [18] Hervé Chaudet, Liliane Pellegrin, Charlotte Gaudin, Gaëtan Texier, Benjamin Queyriaux, Jean-Baptiste Meynard, and Jean-Paul Boutin. A Model-Based Architecture for Supporting Situational Diagnosis in Real-Time Surveillance Systems. *Advances in Disease Surveillance*, 4(152), 2007.
- [19] Hervé Chaudet, Liliane Pellegrin, Jean-Baptiste Meynard, Gaëtan Texier, Olivier Tournebize, Benjamin Queyriaux, and Jean-Paul Boutin. Web services based syndromic surveillance for early warning within French Forces. In *Studies in health technology and informatics*, volume 124, pages 666–671. Stud. Health Tech. Inform., 2006.
- [20] Laurian M Chirica and David F Martin. An Order-Algebraic Definition of Knuthian Semantics. *Mathematical Systems Theory*, 13:1–27, 1979.
- [21] Torkil Clemmensen, Victor Kaptelinin, and Bonnie Nardi. Making HCI theory work: an analysis of the use of activity theory in HCI research. *Behaviour & Information Technology*, 35(8):608–627, 2016.

- [22] Riccardo Cognini, Flavio Corradini, Stefania Gnesi, Andrea Polini, and Barbara Re. Business process flexibility - a systematic literature review with a software systems perspective. *Information Systems Frontiers*, pages 1–29, 2016.
- [23] Duncan L Cooper, G Smith, M Baker, F Chinemana, N Verlander, E Gerard, V Hollyoak, and R Griffiths. National symptom surveillance using calls to a telephone health advice service—United Kingdom, December 2001–February 2003. *MMWR supplements*, 53:179–83, September 2004.
- [24] Bruno Courcelle and Paul Franchi-Zannettacci. Attribute Grammars and Recursive Program Schemes I and II. *Theoretical Computer Science*, 17:163–257, 1982.
- [25] Elio Damaggio, Richard Hull, and Roman Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with Guard – Stage – Milestone lifecycles. *Information Systems Journal*, 38:561–584, 2013.
- [26] Pierre Deransart and Jan Maluszyński. Relating Logic Programs and Attribute Grammars. *J. Log. Program.*, 2(2):119–155, 1985.
- [27] Pierre. Deransart and Jan Maluszyński. *A Grammatical View of Logic Programming*, volume 348 of *Lecture Notes in Computer Science*. MIT Press, 1993.
- [28] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982.
- [29] J.-C. Desenclos, V. Vaillant, E. Delarocque Astagneau, C. Campèse, D. Che, B. Coignard, I. Bonmarin, D. Lévy Bruhl, and H. de Valk. Principles of an outbreak investigation in public health practice. *Médecine et Maladies Infectieuses*, 37(2):77–94, 2007.
- [30] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-intensive Processes: An overview of contemporary approaches. In *Knowledge Intensive Business Processes (KIBP)*, volume 861, pages 33–47. CEUR–WS, 2012.
- [31] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. *Journal on Data Semantics*, 4(1):29–57, 2015.
- [32] Claudio Di Ciccio and Massimo Mecella. A two-step fast algorithm for the automated discovery of declarative workflows. In *2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*, pages 135–142. IEEE, apr 2013.
- [33] Claudio Di Ciccio, Massimo Mecella, Monica Scannapieco, Diego Zardetto, and Tiziana Catarci. MailOfMine – Analyzing Mail Messages for Mining Artful Collaborative Processes. pages 55–81. Springer, Berlin, Heidelberg, 2012.
- [34] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer, 2013.

- [35] Schahram Dustdar. Caramba—A Process-Aware Collaboration System Supporting Ad hoc and Collaborative Processes in Virtual Teams. *Distributed and Parallel Databases*, 15(1):45–66, January 2004.
- [36] Clarence A Ellis and Karim Keddara. ML-DEWS: Modeling Language to Support Dynamic Evolution within Workflow Systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000.
- [37] Rik Eshuis, Richard Hull, Yutian Sun, and Roman Vaculín. Splitting gsm schemas: A framework for outsourcing of declarative artifact systems. *Information Systems*, 46(Supplement C):157 – 187, 2014.
- [38] Dirk Fahland, Daniel Lübke, Jan Mendling, Hajo Reijers, Barbara Weber, Matthias Weidlich, and Stefan Zugal. Declarative versus Imperative Process Modeling Languages: The Issue of Understandability. pages 353–366. Springer, Berlin, Heidelberg, 2009.
- [39] International Society for Disease Surveillance. Final Recommendation: Core Processes and EHR Requirements for Public Health Syndromic Surveillance. Technical report, ISDS, 2011. <http://www.healthsurveillance.org>.
- [40] Martin Fowler. *Domain Specific Languages*. Addison-Wesley Signature Series, 2010.
- [41] Christian Fritz, Richard Hull, and Jianwen Su. Automatic construction of simple artifact-based business processes. In *12th International Conference on Database Theory – (ICDT '09)*, pages 225–238, New York, USA, 2009.
- [42] Denis Gagné and André Trudel. The Temporal Perspective: Expressing Temporal Constraints and Dependencies in Process Models. *Process Models in BPM and Workflow Handbook*, 2008.
- [43] Harald Ganzinger. Increasing Modularity and Language-Independency in Automatically Generated Compilers. *Science of Computer Programming*, 3(3):223–278, 1983.
- [44] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *ACM SIGPLAN Symposium on Compiler Construction*, pages 157–170, 1984.
- [45] Jean Goubault-Larrecq. Well-Founded Recursive Relations. In *Computer Science Logic, Paris, France*, pages 484–498. Springer, Berlin, Heidelberg, 2001. DOI: 10.1007/3-540-44802-0_34.
- [46] Norbert Gronau and Edzard Weber. Management of Knowledge Intensive Business Processes. In *Business Process Management: Second International Conference, (BPM 2004), Potsdam, Germany*, pages 163–178. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-25970-1_11.
- [47] Barbara A Han and John M Drake. Future directions in analytics for infectious disease intelligence. *EMBO reports*, 17(6):785–789, 2016.

- [48] A. Heitmueller, S. Henderson, W. Warburton, A. Elmagarmid, A. S. Pentland, and A. Darzi. Developing Public Policy To Advance The Use Of Big Data In Health Care. *Health Affairs*, 33(9):1523–1530, 2014.
- [49] Loïc Hélouët and Albert Benveniste. Document Based Modeling of Web Services Choreographies using Active XML. In *ICWS*, pages 291–298. IEEE Computer Society, 2010.
- [50] Jean-Michel Hoc. *Supervision et contrôle de processus : la cognition en situation dynamique*. Presses universitaires de Grenoble, 1996.
- [51] Richard Hull. Artifact-Centric Business Process Models: Brief Survey of Research Results and Challenges. In *OTM 2008*, volume 5332 of *Lecture Notes in Computer Science*, pages 1152–1163. Springer, 2008.
- [52] Richard Hull, Elio Damaggio, Fabiana Fournier, Manmohan Gupta, Fenno (Terry) Heath, Stacy Hobson, Mark H Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculín. Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In *Web Services and Formal Methods - (WS-FM 2010)*, Hoboken, NJ, USA, volume 6551 of *Lecture Notes in Computer Science*, pages 1–24. Springer Berlin Heidelberg, 2011.
- [53] Richard Hull, Elio Damaggio, Riccardo De Masellis, Fabiana Fournier, Manmohan Gupta, Fenno Terry Heath, Stacy Hobson, Mark H Linehan, Sridhar Maradugu, Anil Nigam, Piyawadee Sukaviriya, and Roman Vaculín. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011*, pages 51–62. ACM, 2011.
- [54] Richard Hull, Nanjangud C. NC. Narendra, and Anil Nigam. Facilitating Workflow Interoperation Using Artifact-Centric Hubs. In *Service-Oriented Computing, (ICSOC–ServiceWave 2009)*, Stockholm, Sweden, pages 1–18. Springer Berlin Heidelberg, 2009. 10.1007/978-3-642-10383-4_1.
- [55] Thomas Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, FPCA*, volume 274 of *Lecture Notes in Computer Science*, pages 154–173. Springer, 1987.
- [56] Loïc Jossieran, Anne Fouillet, Nadège Caillère, Dominique Brun-Ney, Danièle Ilef, Gilles Brucker, Helena Medeiros, and Pascal Astagneau. Assessment of a Syndromic Surveillance System Based on Morbidity Data: Results from the Oscour Network during a Heat Wave. *PLoS ONE*, 5(8):1–8, 2010. 10.1371/journal.pone.0011984.
- [57] Sandy Kemsley. The Changing Nature of Work : From Structured to Unstructured , from Controlled to Social. In *Proceedings of the 9th International Business Process Management Conference*, 2011.
- [58] M. J. Khoury and J. P. A. Ioannidis. Big data meets public health. *Science*, 346(6213):1054–1055, 2014.

- [59] Paul Klint, Ralf Lämmel, and Chris Verhoef. Toward an engineering discipline for grammarware. *ACM Transaction on Software Engineering Methodologies*, 14(3):331–380, 2005.
- [60] Donald E Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2):127–145, 1968.
- [61] Santhosh Kumaran, Rong Liu, and Frederick Y. Wu. On the duality of information-centric and activity-centric models of business processes. In *Lecture Notes in Computer Science*, 2008.
- [62] Santhosh Kumaran, Prabir Nandi, Terry Heath, Kumar Bhaskaran, and Raja Das. ADoc-Oriented Programming. In *SAINT*, pages 334–343, 2003.
- [63] Vera Künzle and Manfred Reichert. PHILharmonicFlows: towards a framework for object-aware process management. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(4):205–244, 2011.
- [64] Craig Le Clair and Connie Moore. Dynamic Case Management—An Old Idea Catches New Fire. *Forrester Research Inc.*, 2009.
- [65] H P Lehmann, B E Dixon, and H Kharrazi. Public Health and Epidemiology Informatics: Recent Research and Trends in the United States. *Yearbook of medical informatics*, 10(1):199–206, 2015.
- [66] Raanan Lipshitz and Orna Strauss. Coping with Uncertainty: A Naturalistic Decision-Making Analysis. *Organizational Behavior and Human Decision Processes*, 69(2):149–163, 1997.
- [67] Niels Lohmann and Karsten Wolf. Artifact-Centric Choreographies. In *Service-Oriented Computing, San Francisco, CA, USA*, pages 32–46, 2010.
- [68] Joseph Lombardo, Howard Burkom, Eugene Elbert, Steven Magruder, Sheryl Happel Lewis, Wayne Loschen, James Sari, Carol Sniegowski, Richard Wojcik, and Julie Pavlin. A systems overview of the Electronic Surveillance System for the Early Notification of Community-Based Epidemics (ESSENCE II). *Journal of Urban Health*, 80(1):i32–i42, 2003.
- [69] Qingqi Long. A framework for data-driven computational experiments of inter-organizational collaborations in supply chain networks. *Information Sciences*, 399:43–63, 2017.
- [70] Mike Marin, Richard Hull, and Roman Vaculín. Data centric BPM and the emerging case management standard: A short survey. In *Lecture Notes in Business Information Processing*, volume 132 LNBIP, pages 24–30, 2013.
- [71] Mike A. Marin. Introduction to the Case Management Model and Notation (CMMN). 2016. <https://arxiv.org/pdf/1608.05011.pdf>.
- [72] Alberto Martelli and Ugo Montanari. An Efficient Unification Algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

- [73] Brian H Mayoh. Attribute Grammars and Mathematical Semantics. *SIAM J. Comput.*, 10(3):503–518, 1981.
- [74] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344, 2005.
- [75] Giorgio De Michelis and Clarence A Ellis. Computer Supported Cooperative Work and Petri Nets. In *Advanced Course on Petri Nets, Dagstuhl 1996*, volume 1492 of *Lecture Notes in Computer Science*, pages 125–153. Springer, 1998.
- [76] George A Miller. The cognitive revolution: a historical perspective. *Trends in Cognitive Sciences*, 7(3):141–144, 2003.
- [77] Tadao Murata. Petri Nets: Properties, analysis and applications. In *Proceedings of the IEEE*, volume 74, 1989.
- [78] Anil Nigam and Nathan S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42(3):428–445, 2003.
- [79] Robert Fondze Jr Nsaibirni, Eric Badouel, Gaëtan Texier, and Georges-Edouard Kouamou. Active-Workspaces: A Dynamic Collaborative Business Process Model for Disease Surveillance Systems. In *HIMS'16 - The 2nd International Conference on Health Informatics and Medical Systems, Las Vegas, USA*, 2016.
- [80] Robert Fondze Jr Nsaibirni and Gaëtan Texier. User Interactions in Dynamic Processes: Modeling User Interactions in Dynamic Collaborative Processes using Active Workspaces. In *Proceedings of CARI'16*, pages 109–116, Hammamet, Tunisia, 2016.
- [81] Robert Fondze Jr Nsaibirni, Gaëtan Texier, and Georges-Edouard Kouamou. Modelling disease surveillance using Active Workspaces. In *Conference de Recherche en Informatique, Yaoundé, Cameroon*, 2015.
- [82] Robert Fondze Jr Nsaibirni, Gaëtan Texier, Patrice Tchendjou, Georges-Edouard Kouamou, Richard Njouom, Maurice Demanou, and Maurice Tchuenta. An Early Warning Surveillance Platform for Developing Countries. *Online Journal of Public Health Informatics*, 6(1):e10, 2014.
- [83] Object Management Group. Business Process Model and Notation V2.0. Technical report, OMG, 2010. <http://www.bpmn.org/>.
- [84] Donald R Olson, Marc Paladini, William B Lober, David L Buckeridge, and ISDS Distribute Working Group. Applying a New Model for Sharing Population Health Data to National Syndromic Influenza Surveillance: DiSTRIBuTE Project Proof of Concept, 2006 to 2009. *PLoS Currents*, 3:RRN1251, 2011. 10.1371/currents.RRN1251.
- [85] World Health Organization and Centers For Disease Control. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. Technical Report July, WHO/CDC, 2001.

- [86] Jukka Paakki. Attribute Grammar Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.
- [87] George Papamarkos, Alexandra Poulouvasilis, and Peter T. Wood. Event-condition-action rule languages for the semantic web. *Workshop on Semantic Web*, pages 855–864, 2003.
- [88] Fabio Paternò. *Model-based design and evaluation of interactive applications*. Springer, 1st edition, 1999.
- [89] Liliane Pellegrin, Charlotte Gaudin, Nathalie Bonnardel, and Hervé Chaudet. Collaborative Activities During an Outbreak Early Warning Assisted by a Decision-Supported System (ASTER). *International Journal of Human-Computer Interaction*, 26(2-3):262–277, March 2010.
- [90] Liliane Pellegrin, Charlotte Gaudin, Nathalie Bonnardel, Gaëtan Texier, Jean-Baptiste Meynard, and Hervé Chaudet. Formalizing collaboration in decision-making : a case study in military epidemiological early warning. In *Proceedings of the 9th Bi-annual International Conference on Naturalistic Decision Making*, NDM'09, pages 40–40, Swindon, UK, 2009. BCS Learning & Development Ltd.
- [91] Liliane Pellegrin, Charlotte Gaudin, Gaëtan Texier, Jean-Baptiste Meynard, and Hervé Chaudet. Near real-time outbreak surveillance system for early warning as a JCS. In *Annual European Conference on Cognitive Ergonomics - ECCE '10*, pages 33–40, New York, New York, USA, 2010. ACM Press.
- [92] M. Pesic, M. H. Schonenberg, N. Sidorova, and Wil M. P. van der Aalst. Constraint-Based Workflow Models: Change Made Easy. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems, Part I*, pages 77–94. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-76848-7_7.
- [93] Stefanie Rinderle, Manfred Reichert, and Peter Dadam. Correctness criteria for dynamic changes in workflow systems - A survey. In *Data and Knowledge Engineering*, volume 50, pages 9–34. North-Holland, 2004.
- [94] Henry M. Robert, Sarah Corbin Robert, Henry M. Robert III, William J. Evans, Daniel H. Honemann, Thomas J. Balch, Daniel E. Seabold, and Shmuel Gerber. *Robert's rules of order newly revised*. Da Capo Press, 11th edition, 2011.
- [95] Yvonne Rogers. HCI Theory: Classical, Modern, and Contemporary. *Synthesis Lectures on Human-Centered Informatics*, 5(2):1–129, 2012.
- [96] João Saraiva and S. Doaitse Swierstra. Generating Spreadsheet-Like Tools from Strong Attribute Grammars. In *Generative Programming and Component Engineering, GPCE 2003*, volume 2830 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2003.
- [97] João Saraiva, S. Doaitse Swierstra, and Matthijs F Kuiper. Functional Incremental Attribute Evaluation. In *Compiler Construction, CC 2000*, volume 1781 of *Lecture Notes in Computer Science*, pages 279–294. Springer Berlin Heidelberg, 2000.

- [98] Helen Schonenberg, Ronny Mans, Nick Russell, Nataliya Mulyar, and Wil van der Alst. Process Flexibility: A survey of contemporary approaches. *Lecture Notes in Business Information Processing*, 10(Part I):16–30, 2008.
- [99] Benyun Shi, Shang Xia, and Jiming Liu. A Complex Systems Approach to Infectious Disease Surveillance and Response. pages 524–535. Springer International Publishing, Cham, 2013. DOI: 10.1007/978-3-319-02753-1_53.
- [100] Marlow Simon. Haskell 2010 Language Report. Technical report, 2010. <https://www.haskell.org/>.
- [101] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.
- [102] Neville A. Stanton. Hierarchical task analysis: Developments, applications, and extensions. *Applied Ergonomics*, 37(1):55–79, 2006.
- [103] J. K. Strosnider, P. Nandi, S. Kumaran, S. Ghosh, and A. Arsanjani. Model-driven synthesis of SOA solutions. *IBM Systems Journal*, 47(3):451–432, 2008.
- [104] Keith D. Swenson. *Mastering the Unpredictable: How Adaptive Case Management Will Revolutionize the Way That Knowledge Workers Get Things Done*. Meghan-Kiffer Press, 2010.
- [105] S Doaitse Swierstra, Pablo R Azero Alcocer, and João Saraiva. Designing and Implementing Combinator Languages. In *Advanced Functional Programming*, pages 150–206. Springer Berlin Heidelberg, 1998.
- [106] S. Doaitse Swierstra and Harald Vogt. Higher Order Attribute Grammars. In *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer, 1991.
- [107] Gaëtan Texier. *Méthode d'évaluation des algorithmes de détection temporelle des épidémies dans le cadre de l'alerte précoce*. PhD Thesis, Aix-Marseille Université, 2016.
- [108] Gaëtan Texier and Y. Buisson. From epidemic outbreak detection to anticipation. *Revue d'Epidemiologie et de Sante Publique*, 2010.
- [109] Gaëtan Texier, Liliane Pellegrin, Claire Vignal, Jean-Baptiste Meynard, Xavier Deparis, and Hervé Chaudet. Dealing with uncertainty when using a surveillance system. *International Journal of Medical Informatics*, 104:65–73, 2017.
- [110] Triple S Project. Assessment of syndromic surveillance in Europe. *The Lancet*, 378(9806):1833–1834, 2011. doi:10.1016/S0140-6736(11)60834-9.
- [111] Roman Vaculín, Richard Hull, Terry Heath, Craig Cochran, Anil Nigam, and Piyawadee Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, pages 151–160, 2011.
- [112] Wil M. P. van Der Aalst. the Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.

- [113] Wil M P van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [114] Wil M. P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.
- [115] Wil M P van der Aalst. Process Discovery: An Introduction. In *Process Mining*, pages 163–194. Springer Berlin Heidelberg, Berlin, Heidelberg, 2nd edition, 2016.
- [116] Wil M P van der Aalst. Process Mining: The Missing Link. In *Process Mining*, pages 25–52. Springer Berlin Heidelberg, Berlin, Heidelberg, 2nd edition, 2016.
- [117] Wil M P van Der Aalst and a. H M Ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [118] Wil M P van der Aalst, A. H M Ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
- [119] Wil. M P Van Der Aalst, Arthur H. M. ter Hofstede, and Mathias Weske. Business Process Management : A Survey. In *Business Process Management: International Conference, (BPM 2003) Eindhoven, The Netherlands*, pages 1–12. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. DOI: 10.1007/3-540-44895-0_1.
- [120] Wil. M P Van Der Aalst, Mathias Weske, and Dolf Gränbauer. Case handling : a new paradigm for business process support. *Data & Knowledge Engineering*, 53(2):129–162, 2005.
- [121] Wil M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999.
- [122] Harald Vogt, S. Doaitse Swierstra, and Matthijs F Kuiper. Higher-Order Attribute Grammars. In *PLDI*, pages 131–145. ACM, 1989.
- [123] Garrick Wallstrom, Brigham Anderson, and Others. *Handbook of Biosurveillance*. Elsevier, 2006. DOI: 10.1016/B978-012369378-5/50014-4.
- [124] Mathias Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer Berlin Heidelberg, 2nd edition, 2012. DOI: 10.1007/978-3-642-28616-2.
- [125] Jan Willem Klob and Roel de Vrijer. First-order term rewriting systems. In *Term rewriting systems*, pages 24–58. Cambridge University Press, 2005.
- [126] M. D. Wilson, P. J. Barnard, T. R. G. Green, and A. Maclean. Knowledge-based task analysis for human-computer systems. *Working with Computers: Theory versus Outcome*, pages 47–87, 1987.
- [127] T. Winograd and F. Flores. *Understanding Computer and Cognition: A new Foundation for Design*. Norwood, 1986.
- [128] World Health Organization. Communicable disease surveillance and response systems - Guide to monitoring and evaluating. Technical report, WHO, 2006.

- [129] World Health Organization. *WHO | International Health Regulations (2005)*. 2 edition, 2008.
- [130] Eric Van Wyk. Implementing aspect-oriented programming constructs as modular language extensions. *Science of Computer Programming*, 68(1):38–61, 2007.
- [131] Eric Van Wyk, Oege de Moor, Kevin Backhouse, and Paul Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. In *Compiler Construction, ETAPS 2002, Grenoble, France*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2002.
- [132] Yan Xiao. Artifacts and collaborative work in healthcare: methodological, theoretical, and technological implications of the tangible. *Journal of Biomedical Informatics*, 38(1):26–33, 2005.
- [133] W A Yasnoff, P W O’Carroll, D Koo, R W Linkins, and E M Kilbourne. Public health informatics: improving and transforming public health in the information age. *Journal of public health management and practice : JPHMP*, 6(6):67–75, 2000.
- [134] Daniel. Zeng. *Infectious disease informatics and biosurveillance*. Springer, 2011.
- [135] Xiang Zhao, Yuriy Brun, and Leon J. Osterweil. Supporting process undo and redo in software engineering decision making. In *International Conference on Software and System Process, ICSSP 2013*, pages 56–60, 2013.

ANNEX
List Of Publications

Active Workspaces: Distributed Collaborative Systems based on Guarded Attribute Grammars

Eric Badouel
Inria and LIRIMA
Campus de Beaulieu
35042 Rennes, France
eric.badouel@inria.fr

Loïc Hélouët
Inria
Campus de Beaulieu
35042 Rennes, France
loic.helouet@inria.fr

Georges-Edouard
Kouamou
ENSP and LIRIMA
BP 8390, Yaoundé, Cameroon
georges.kouamou@lirima.org

Christophe Morvan
Université Paris-Est
UPEMLV, F-77454
Marne-la-Vallée, France
christophe.morvan@u-pem.fr

Robert Fondze Jr
Nsaibirni
UY1 and LIRIMA
BP 812, Yaoundé, Cameroon
nsairobby@gmail.com

ABSTRACT

This paper presents a purely declarative approach to artifact-centric collaborative systems, a model which we introduce in two stages. First, we assume that the workspace of a user is given by a mindmap, shortened to a map, which is a tree used to visualize and organize tasks in which he or she is involved, with the information used for the resolution of these tasks. We introduce a model of *guarded attribute grammar*, or GAG, to help the automation of updating such a map. A GAG consists of an underlying grammar, that specifies the logical structure of the map, with semantic rules which are used both to govern the evolution of the tree structure (how an open node may be refined to a subtree) and to compute the value of some of its attributes (which derives from contextual information). The map enriched with this extra information is termed an *active workspace*. Second, we define collaborative systems by making the various user's active workspaces communicate with each other. The communication uses message passing without shared memory thus enabling convenient distribution on an asynchronous architecture. We present some formal properties of the model of guarded attribute grammars, then a language for their specification and we illustrate the approach on a case study for a disease surveillance system.

CCS Concepts

•Information systems → Enterprise information systems; Collaborative and social computing systems and tools; Asynchronous editors; •Theory of computation → Interactive computation; Distributed computing models; Grammars and context-free languages; •Software and its engineering → Specification languages;

Keywords

Business Artifacts, Case Management, Attribute Grammars

Copyright is held by the authors. This work is based on an earlier work: SAC'15 Proceedings of the 2015 ACM Symposium on Applied Computing, Copyright 2015 ACM 978-1-4503-3196-8. <http://dx.doi.org/10.1145/2695664.2695698>

1. INTRODUCTION

This paper presents a modular and purely declarative model of artifact-centric collaborative systems, which is user-centric, easily distributable on an asynchronous architecture and re-configurable without resorting to global synchronizations.

Traditional Case Management Systems rely on workflow models. The emphasis is put on the orchestration of activities involving humans (the *stakeholders*) and software systems, in order to achieve some global objective. In this context, stress is often put on control and coordination of tasks required for the realization of a particular service. Such systems are usually modeled with centralized and state-based formalisms like automata, Petri nets or statecharts. They can also directly be specified with dedicated notations like BPEL [33] or BPMN [22].

One drawback of existing workflow formalisms is that *data* exchanged during the processing of a task play a secondary role when not simply ignored. However, data can be tightly connected with control flows and should not be overlooked. Actually, data contained in a request may influence its processing. Conversely different decisions during the treatment of a case may produce distinct output-values.

Similarly, stakeholders are frequently considered as second class citizens in workflow systems: They are modeled as plain resources, performing specific tasks for a particular case, like machines in assembly lines. As a result, workflow systems are ideal to model fixed production schemes in manufactures or organizations, but can be too rigid to model open architectures where the evolving rules and data require more flexibility.

Data-centric workflow systems were proposed by IBM [32, 24, 10]. They put stress on the exchanged documents, the so-called *Business Artifacts*, also known as *business entities with lifecycles*. An artifact is a document that conveys all the information concerning a particular case from its inception in the system until its completion. It contains all the relevant information about the entity together with a lifecycle that models its possible evolutions through the business

process. Several variants presenting the life cycle of an artifact by an automaton, a Petri net [29], or logical formulas depicting legal successors of a state [10] have been proposed. However, even these variants remain state-based centralized models in which stakeholders do not play a central role.

Guard-Stage-Milestone (GSM), a declarative model of the lifecycle of artifacts was recently introduced in [25, 11]. This model defines *Guards*, *Stages* and *Milestones* to control the enabling, enactment and completion of (possibly hierarchical) activities. The GSM lifecycle meta-model has been adopted as a basis of the OMG standard *Case Management Model and Notation* (CMMN). The GSM model allows for dynamic creation of subtasks (the *stages*), and handles data attributes. Furthermore, guards and milestones attached to stages provide declarative descriptions of tasks inception and termination. However, interaction with users are modeled as incoming messages from the environment, or as events from low-level (atomic) stages. In this way, users do not contribute to the choice of a workflow for a process. The semantics of GSM models is given in terms of global snapshots. Events can be handled by all stages as soon as they are produced, and guard of a stage can refer to attributes of distant stages. Thus this model is not directly executable on a distributed architecture. As highlighted in [18], distributed implementation may require restructuring the original GSM schema and relies on locking protocols to ensure that the outcome of the global execution is preserved.

This paper presents a declarative model for the specification of collaborative systems where the stakeholders interact according to an asynchronous message-based communication schema.

Case-management usually consists in assembling relevant information by calling *tasks*, which may in turn call subtasks. Case elicitation needs not be implemented as a sequence of successive calls to subtasks, and several subtasks can be performed in parallel. To allow as much concurrency as possible in the execution of tasks, we favor a *declarative* approach where task dependencies are specified without imposing a particular execution order.

Attribute grammars [28, 34] are particularly adapted to that purpose. The model proposed in this paper is a variant of attribute grammar, called *Guarded Attributed Grammar* (GAG). We use a notation reminiscent of unification grammars, and inspired by the work of Deransart and Maluszynski [15] relating attribute grammars with definite clause programs.

A production of a grammar is, as usual, described by a left-hand side, indicating a non-terminal to expand, and a right-hand side, describing how to expand this non-terminal. We furthermore interpret a production of the grammar as a way to decompose a task (the symbol in the left-hand side of the production) into sub-tasks associated with the symbols in its right-hand side. The semantics rules basically serve as a glue between the task and its sub-tasks by making the necessary connections between the corresponding inputs and outputs (associated respectively with inherited and synthesized attributes).

In this declarative model, the lifecycle of artifacts is left implicit. Artifacts under evaluation can be seen as incom-

plete structured documents, i.e., trees with *open nodes* corresponding to parts of the document that remain to be completed. Each open node is attached a so-called *form* interpreted as a task. A form consists of a task name together with some inherited attributes (data resulting from previous executions) and some synthesized attributes. The latter are variables subscribing to the values that will emerge from task execution.

Productions are *guarded* by patterns occurring at the inherited positions of the left-hand side symbol. Thus a production is enabled at an open node if the patterns match with the corresponding attribute values as given in the form. The evolution of the artifact thus depends both on previously computed data (stating which production is enabled) and the stakeholder's decisions (choosing a particular production amongst those which are enabled at a given moment, and inputting associated data). Thus GAGs are both *data-driven* and *user-centric*.

Data manipulated in guarded attributed grammars are of two kinds. First, the tasks communicate using *forms* which are temporary information used for communication purpose only, essentially for requesting values. Second, *artifacts* are structured documents that record the history of cases (log of the system). An artifact grows monotonically –we never erase information. Moreover, every part of the artifact is edited by a unique stakeholder –the owner of the corresponding nodes– hence avoiding edition conflicts. These properties are instrumental to obtain a simple and robust model that can easily be implemented on a distributed asynchronous architecture.

The modeling of a distributed collaborative system using GAG proceeds in two stages. First, we represent the workspaces of the various stakeholders as the collections of the artifacts they respectively handle. An artifact is a structured document with some active parts. Indeed, an open node is associated with a task that implicitly describes the data to be further substituted to the node. For that reason these workspaces are termed *active workspaces*. Second, we define collaborative systems by making the various user's active workspaces communicate with each other using asynchronous message passing.

This notion of *active documents* is close to the model of Active XML introduced by Abiteboul et al. [2] which consists of semi-structured documents with embedded service calls. Such an embedded service call is a query on another document, triggered when a corresponding guard is satisfied. The model of active documents can be distributed over a network of machines [1, 23]. This setting can be instanced in many ways, according to the formalism used for specifying the guards, the query language, and the class of documents. The model of guarded attribute grammars is close to this general schema with some differences: First of all, guards in GAGs apply to the attributes of a single node while guards in AXML are properties that can be checked on a complete document. The invocation of a service in AXML creates a temporary document (called the workspace) that is removed from the document when the service call returns. In GAGs, a rule applied to solve a task adds new children to the node, and all computations performed for a task are preserved in the artifact. This provides a kind of monotony to artifacts,

an useful property for verification purpose.

The rest of the paper is organized as follows. Section 2 introduces the model of guarded attribute grammars and focuses on their use to standalone applications. The approach is extended in Section 3 to account for systems that call for external services. In this context we introduce a composition of guarded attribute grammars. Some formal properties of guarded attribute grammars are studied in Section 4. Section 5 presents some notations and constructions allowing us to cope with the complexity of real-life systems. This specification language is illustrated in Section 6 on a case study for a disease surveillance system. Finally an assessment of the model and future research directions are given in conclusion.

2. GUARDED ATTRIBUTE GRAMMARS

This section is devoted to a presentation of the model of guarded attribute grammars. We start with an informal presentation that shows how the rules of a guarded attribute grammar can be used to structure the workspace of a stakeholder and formalize the workspace update operations. In two subsequent subsections we respectively define the syntax and the behavior of guarded attribute grammars. The section ends with basic examples used to illustrate some of the fundamental characteristics of the model.

2.1 A Grammatical Approach to Active Workspaces

Our model of collaborative systems is centered on the notion of user’s workspace. We assume that the workspace of a user is given by a mindmap –simply call a *map* hereafter. It is a tree used to visualize and organize tasks in which the user is involved together with information used for the resolution of the tasks. The workspace of a given user may, in fact, consist of several maps where each map is associated with a particular *service* offered by the user. To simplify, one can assume that a user offers a unique service so that any workspace can be identified with its graphical representation as a map.

For instance the map shown in Fig. 1 might represent the workspace of a clinician acting in the context of a disease surveillance system.

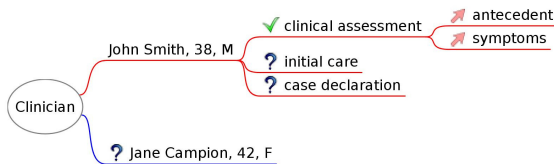


Figure 1: Active workspace of a clinician

The service provided by the clinician is to identify the symptoms of influenza in a patient, clinically examines the patient

eventually placing him under therapeutic care, declaring the suspect cases to the Disease Surveillance Center (DCS), and monitoring the patient based on subsequent requests from the epidemiologist or the biologist.

Each call to this service, namely when a new patient comes to the clinician, creates a new tree rooted at the central node of the map. This tree is an *artifact* that represents a structured document for recording information about the patient all along being taken over in the system. Initially the artifact is reduced to a single (open) node that bears information about the name, age and sex of the patient. An open node, graphically identified by a question mark, represents a *pending task* that requires the attention of the clinician. In our example the initial task of an artifact is to clinically examine the patient. This task is refined into three subtasks: clinical assessment, initial care, and case declaration.

Our first goal is to ease the work of the clinician by avoiding a manual updating of the map. In order to automate transformations of the map we must first proceed to a classification of the different nodes –indicating their *sort*. Intuitively two open nodes are of the same sort when they can be refined by the same subtrees –they can have the same future. It then becomes possible, depending on the sort of an open node, to associate with it specific information –the *attributes* of the sort– and to specify in which way the node can be developed.

We interpret a task as a problem to be solved, that can be completed by refining it into sub-tasks using *business rules*. In a first approximation, a (business) rule can be modelled by a *production* $P : s_0 \rightarrow s_1 \dots s_n$ expressing that task s_0 can be reduced to subtasks s_1 to s_n . For instance the production

$$\text{patient} \rightarrow \begin{array}{l} \text{clinical_assessment} \\ \text{initial_care} \\ \text{case_declaration} \end{array}$$

states that a task of sort **patient**, the axiom of the grammar associated with the service provided by the clinician, can be refined by three subtasks whose sorts are respectively **clinical_assessment**, **initial_care**, and **case_declaration**.

If several productions with the same left-hand side s_0 exist then the choice of a particular production corresponds to a *decision* made by the user. For instance the clinician has to decide whether the case under investigation has to be declared to the Disease Surveillance Center or not. This decision can be reflected by the following two productions:

$$\begin{array}{l} \text{suspect_case} : \text{case_declaration} \rightarrow \text{follow_up} \\ \text{benign_case} : \text{case_declaration} \rightarrow \end{array}$$

If the case is reported as suspect then the clinician will have to follow up the case according to further requests of the biologist or of the epidemiologist. On the contrary, if the clinician has described the case as benign, it is closed with no follow up actions, More generally the tasks on the right-hand side of each production represent what remains to be done to resolve the task on the left-hand side in case this production is chosen.

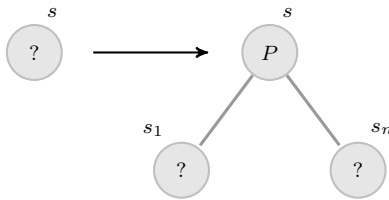
If P is the unique production having s_0 in its left-hand side,

then there is no real decision to make and such a rule is interpreted as a logical decomposition of the task s_0 into subtasks s_1 to s_n . Such a production will automatically be triggered without human intervention.

Accordingly, we model an artifact as a tree whose nodes are sorted. We write $X :: s$ to indicate that node X is of sort s . An artifact is given by a set of equations of the form $X = P(X_1, \dots, X_n)$, stating that $X :: s$ is a node labeled by production $P : s \rightarrow s_1 \dots s_n$ and with successor nodes $X_1 :: s_1$ to $X_n :: s_n$. In that case node X is said to be a *closed* node defined by equation $X = P(X_1, \dots, X_n)$ – we henceforth assume that we do not have two equations with the same left-hand side. A node $X :: s$ defined by no equation (i.e. that appears only in the right hand side of an equation) is an *open* node. It corresponds to a pending task of sort s .

The lifecycle of an artifact is implicitly given by the set of productions:

1. The artifact initially associated with a case is reduced to a single open node.
2. An open node X of sort s can be *refined* by choosing a production $P : s \rightarrow s_1 \dots s_n$ that fits its sort. The open node X becomes a closed node –defined as $X = P(X_1, \dots, X_n)$ – under the decision of applying production P to it. In doing so task s associated with X is replaced by n subtasks s_1 to s_n and new open nodes $X_1 :: s_1$ to $X_n :: s_n$ are created accordingly.



3. The case has reached completion when its associated artifact is closed, i.e. it no longer contains open nodes.

Using the productions, the stakeholder can edit his workspace –the map– by selecting an open node –a pending task–, choosing one of the business rules that can apply to it, and inputting some values –information transmitted to the system.

However, plain context-free grammars are not sufficient to model the interactions and data exchanged between the various tasks associated with open nodes. For that purpose, we attach additional information to open nodes using *attributes*. Each sort $s \in S$ comes equipped with a set of *inherited* attributes and a set of *synthesized* attributes. Values of attributes are given by *terms* over a ranked alphabet. Recall that such a term is either a variable or an expression of the form $c(t_1, \dots, t_n)$ where c is a symbol of rank n , and t_1, \dots, t_n are terms. In particular a constant c , i.e. a symbol of rank 0, will be identified with the term $c()$. We denote by $var(t)$ the set of variables used in term t .

DEFINITION 2.1 (FORMS). A **form** of sort s is an expression $F = s(t_1, \dots, t_n)\langle u_1, \dots, u_m \rangle$ where t_1, \dots, t_n (respectively u_1, \dots, u_m) are terms over a ranked alphabet – the alphabet of attribute’s values– and a set of variables $var(F)$. Terms t_1, \dots, t_n give the values of the **inherited attributes** and u_1, \dots, u_m the values of the **synthesized attributes** attached to form F .

(Business) rules are productions where sorts are replaced by forms of the corresponding sorts. More precisely, a rule is of the form

$$s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle \rightarrow s_1(t_1^{(1)}, \dots, t_{n_1}^{(1)})\langle y_1^{(1)}, \dots, y_{m_1}^{(1)} \rangle$$

$$\vdots$$

$$s_k(t_1^{(k)}, \dots, t_{n_k}^{(k)})\langle y_1^{(k)}, \dots, y_{m_k}^{(k)} \rangle$$

where the p_i ’s, the u_j ’s, and the $t_j^{(\ell)}$ ’s are terms and the $y_j^{(\ell)}$ ’s are variables. The forms in the right-hand side of a rule are *tasks* given by forms

$$F = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$$

where the synthesized positions are (distinct) variables y_1, \dots, y_m –i.e., they are not instantiated. The rationale is that we invoke a task by filling in the inherited positions of the form –the entries– and by indicating the variables that expect to receive the results returned during task execution –the *subscriptions*.

Any open node is thus attached to a task. The corresponding task execution is supposed (i) to construct the tree that will refine the open node and (ii) to compute the values of the synthesized attributes –i.e., it should return the subscribed values. A task is enacted by applying rules. More precisely, the rule can apply in an open node X when its left-hand side matches with task $s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ attached to node X . For that purpose the terms p_i ’s are used as patterns that should match the corresponding data d_i ’s. When the rules applies, new open nodes are created and they are respectively associated with the forms –tasks– in the right-hand side of the rule. The values of u_j ’s are then returned to the corresponding variables y_j ’s that subscribed to these values. For instance applying rule (see Fig. 2)

$$R : s_0(a(x_1, x_2))\langle b(y'_1), y'_2 \rangle \rightarrow s_1(c(x_1))\langle y'_1 \rangle \quad s_2(x_2, y'_1)\langle y'_2 \rangle$$

to a node associated with tasks $s_0(a(t_1, t_2))\langle y_1, y_2 \rangle$ gives rise to the substitution $x_1 = t_1$ and $x_2 = t_2$. The two newly-created open nodes are respectively associated with the tasks $s_1(c(t_1))\langle y'_1 \rangle$ and $s_2(t_2, y'_1)\langle y'_2 \rangle$ and the values $b(y'_1)$ and y'_2 are substituted to the variables y_1 and y_2 respectively.

This formalism puts emphasis on a declarative (logical) decomposition of tasks to avoid overconstrained schedules. Indeed, semantic rules and guards do not prescribe any ordering on task executions. Moreover ordering of tasks depend on the exchanged data and therefore are determined at run time. In this way, the model allows as much concurrency as possible in the execution of the current pending tasks.

Furthermore the model can incrementally be designed by observing user’s daily practice and discussing with her: We can initially let the user manually develops large parts of the map and progressively improve the automation of the process by refining the classification of the nodes –improving

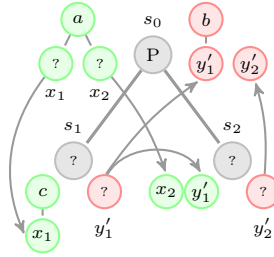


Figure 2: A business rule

our knowledge on the ontology of the system— and introducing new business rules when recurrent patterns of activities are detected.

2.2 Guarded Attribute Grammars Syntax

Attribute grammars, introduced by Donald Knuth in the late sixties [28], have been instrumental in the development of syntax-directed transformations and compiler design. More recently this model has been revived for the specification of structured document’s manipulations mainly in the context of web-based applications. The expression *grammareware* has been coined in [27] to qualify tools for the design and customization of grammars and grammar-dependent softwares. One such tool is the UUAG system developed by Swierstra and his group. They relied on purely functional implementations of attribute grammars [26, 36, 5] to build a domain specific languages (DSL) as a set of functional combinators derived from the semantic rules of an attribute grammar [16, 36, 35].

An attribute grammar is obtained from an underlying grammar by associating each sort s with a set $Att(s)$ of *attributes*—which henceforth should exist for each node of the given sort— and by associating each production $P : s \rightarrow s_1 \dots s_n$ with semantic rules describing the functional dependencies between the attributes of a node labelled P (hence of sort s) and the attributes of its successor nodes—of respective sorts s_1 to s_n . We use a non-standard notation for attribute grammars, inspired from [14, 15]. Let us introduce this notation on an example before proceeding to the formal definition.

EXAMPLE 2.2 (FLATTENING OF A BINARY TREE).

Our first illustration is the classical example of the attribute grammar that computes the flattening of a binary tree, i.e., the sequence of the leaves read from left to right. The semantic rules are usually presented as shown in Fig. 2.2. The sort bin of binary trees has two attributes: The inherited attribute h contains an accumulating parameter and the synthesized attribute s eventually contains the list of leaves of the tree appended to the accumulating parameter. Which we may write as $t \cdot s = flatten(t) ++ t \cdot h$, i.e., $t \cdot s = flat(t, t \cdot h)$ where $flat(t, h) = flatten(t) ++ h$. The semantics rules stem from the identities:

$$\begin{aligned} flatten(t) &= flat(t, Nil) \\ flat(Fork(t_1, t_2), h) &= flat(t_1, flat(t_2, h)) \\ flat(Leaf_a, h) &= Cons_a(h) \end{aligned}$$

We present the semantics rules of Fig. 2.2 using the following

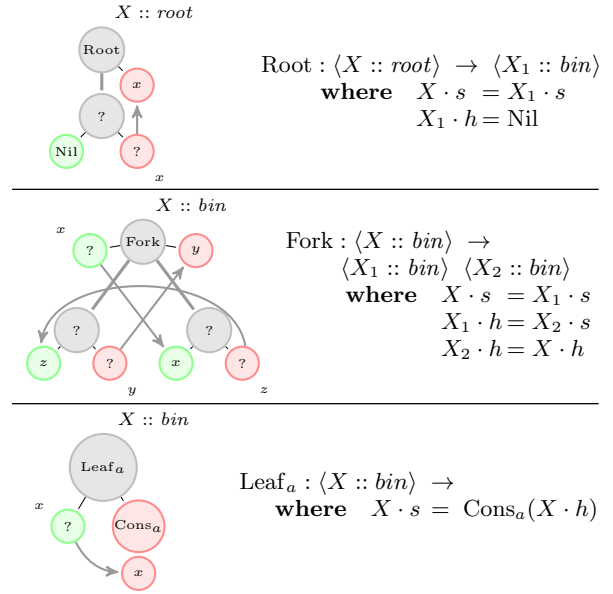


Figure 3: Flattening of a binary tree

syntax:

$$\begin{aligned} \text{Root} & : \quad root() \langle x \rangle \rightarrow bin(Nil) \langle x \rangle \\ \text{Fork} & : \quad bin(x) \langle y \rangle \rightarrow bin(z) \langle y \rangle bin(x) \langle z \rangle \\ \text{Leaf}_a & : \quad bin(x) \langle Cons_a(x) \rangle \rightarrow \end{aligned}$$

The *syntactic categories* of the grammar, also called its *sorts*, namely $root$ and bin are associated with their inherited attributes—given as a list of arguments: (t_1, \dots, t_n) — and their synthesized attributes—the co-arguments: (u_1, \dots, u_m) . A variable x is an *input variable*, denoted as $x^?$, if it appears in an inherited attribute in the left-hand side or in a synthesized attribute in the right-hand side. It corresponds to a piece of information stemming respectively from the context of the node or from the subtree rooted at the corresponding successor node. These variables should be pairwise distinct. Symmetrically a variable is an *output variable*, denoted as $x^!$, if it appears in a synthesized attribute of the left-hand side or in an inherited attribute of the right-hand side. It corresponds to values computed by the semantic rules and sent respectively to the context of the node or the subtree rooted at the corresponding successor node. Indeed, if we annotate the occurrences of variables with their polarity—input or output— one obtains:

$$\begin{aligned} \text{Root} & : \quad root() \langle x^! \rangle \rightarrow bin(Nil) \langle x^? \rangle \\ \text{Fork} & : \quad bin(x^?) \langle y^! \rangle \rightarrow bin(z^!) \langle y^? \rangle bin(x^!) \langle z^? \rangle \\ \text{Leaf}_a & : \quad bin(x^?) \langle Cons_a(x^!) \rangle \rightarrow \end{aligned}$$

And if we draw an arrow from the (unique) occurrence of $x^?$ to the (various) occurrences of $x^!$ for each variable x to witness the data dependencies then the above rules correspond precisely to the three figures shown on the left-hand side of Table 2.2. End of Exple 2.2

Guarded attribute grammars extend the traditional model of attribute grammars by allowing patterns rather than plain

variables –as it was the case in the above example– to represent the inherited attributes in the left-hand side of a rule. Patterns allow the semantic rules to process by case analysis based on the shape of some of the inherited attributes, and in this way to handle the interplay between the data –contained in the inherited attributes– and the control –the enabling of rules.

DEFINITION 2.3 (GUARDED ATTRIBUTE GRAMMARS). *Given a set of sorts S with fixed inherited and synthesized attributes, a **guarded attribute grammar** (GAG) is a set of rules $R : F_0 \rightarrow F_1 \cdots F_k$ where the $F_i :: s_i$ are forms. A sort is **used** (respectively **defined**) if it appears in the right-hand side (resp. the left-hand side) of some rule. A guarded attribute grammar G comes with a specific set of sorts **axioms**(G) $\subseteq \text{def}(G) \setminus \text{Use}(G)$ –called the **axioms** of G – that are defined and not used. They are interpreted as the provided services. Sorts which are used but not defined are interpreted as external services used by the guarded attribute grammar. The values of the inherited attributes of left-hand side F_0 are called the **patterns** of the rule. The values of synthesized attributes in the right-hand side are variables. These occurrences of variables together with the variables occurring in the patterns are called the **input occurrences** of variables. We assume that each variable has at most one input occurrence.*

REMARK 2.4. We have assumed in Def. 2.3 that axioms do not appear in the right-hand side of rules. This property will be instrumental to prove that strong-acyclicity –a property that guarantee a safe distribution of the GAG on an asynchronous architecture– can be compositionally verified. Nonetheless a specification that does not satisfy this property can easily be transformed into an equivalent specification that satisfies it: For each axiom s that occurs in the right-hand side of the rule we add a new symbol s' that becomes axioms in the place of s and we add copies of the rules associated with s –containing s in their left-hand side– in which we replace the occurrence of s in the left-hand side by s' . In this way we distinguish s used as a service by the environment of the GAG –role which is now played by s' – from its uses as an internal subtask –role played by s in the transformed GAG.

End of Remark 2.4

A rule of a GAG specifies the values at output positions –value of a synthesized attribute of s_0 or of an inherited attribute of s_1, \dots, s_n . We refer to these correspondences as the **semantic rules**. More precisely, the inputs are associated with (distinct) variables and the value of each output is given by a term.

A variable can have several occurrences. First it may appear (once) as an input and it may also appear in output values. The corresponding occurrence is respectively said to be in an *input* or in an *output position*. One can define the following transformation on rules whose effect is to annotate each occurrence of a variable so that $x^?$ (respectively $x^!$) stands for an occurrence of x in an input position (resp.

in an output position).

$$\begin{aligned} !(F_0 \rightarrow F_1 \cdots F_k) &= ?(F_0) \rightarrow !(F_1) \cdots !(F_k) \\ ?(s(t_1, \dots t_n)(u_1, \dots u_m)) &= s(? (t_1), \dots ? (t_n)) (! (u_1), \dots ! (u_m)) \\ !(s(t_1, \dots t_n)(u_1, \dots u_m)) &= s(! (t_1), \dots ! (t_n)) (? (u_1), \dots ? (u_m)) \\ ?(c(t_1, \dots t_n)) &= c(? (t_1), \dots ? (t_n)) \\ !(c(t_1, \dots t_n)) &= c(! (t_1), \dots ! (t_n)) \\ ?(x) &= x^? \\ !(x) &= x^! \end{aligned}$$

The conditions stated in Definition 2.3 say that in the labelled version of a rule each variable occurs at most once in an input position, i.e., that $\{?(F_0), !(F_1), \dots, !(F_k)\}$ is an admissible labelling of the set of forms in rule R according to the following definition.

DEFINITION 2.5 (LINK GRAPH). *A labelling in $\{?, !\}$ of the variables $\text{var}(\mathcal{F})$ of a set of forms \mathcal{F} is **admissible** if the labelled version of a form $F \in \mathcal{F}$ is given by either $!F$ or $?F$ and each variable has at most one occurrence labelled with $?$. The occurrence $x^?$ identifies the place where the value of variable x is defined and the occurrences of $x^!$ identify the places where this value is used. The **link graph** associated with an admissible labelling of a set of forms \mathcal{F} is the directed graph whose vertices are the occurrences of variables with an arc from v_1 to v_2 if these vertices are occurrences of a same variable x , labelled $?$ in v_1 and $!$ in v_2 . This arc, depicted as follows,*



*means that the value produced in the **source vertex** v_1 should be forwarded to the **target vertex** v_2 . Such an arc is called a **data link**.*

DEFINITION 2.6 (UNDERLYING GRAMMAR). *The **underlying grammar** of a guarded attribute grammar G is the context-free grammar $\mathcal{U}(G) = (N, T, A, \mathcal{P})$ where*

- the non-terminal symbols $s \in N$ are the defined sorts,
- $T = S \setminus N$ is the set of terminal symbols –the external services–,
- $A = \text{axioms}(G)$ is the set of axioms of the guarded attribute grammar, and
- the set of productions \mathcal{P} is made of the underlying productions $\mathcal{U}(R) : s_0 \rightarrow s_1 \cdots s_k$ of rules $R : F_0 \rightarrow F_1 \cdots F_k$ with $F_i :: s_i$.

*A guarded attribute grammar is said to be **autonomous** when its underlying grammar contains no terminal symbols.*

Intuitively an autonomous guarded attribute grammar represents a standalone application: It corresponds to the description of particular services, associated with the axioms, whose realizations do not rely on external services.

2.3 The Behavior of Autonomous Guarded Attribute Grammars

Attribute grammars are applied to input abstract syntax trees. These trees are usually produced by some parsing

algorithm during a previous stage. The semantic rules are then used to decorate the node of the input tree by attribute values. In our setting, the generation of the tree and its evaluation using the semantic rules are intertwined since the input tree represents an artifact under construction. An artifact is thus an incomplete abstract syntax tree that contains closed and open nodes. A closed node is labelled by the rule that was used to create it. An open node is associated with a form that contains all the needed information for its further refinements. The information attached to an open node consists of the sort of the node and the current value of its attributes. The synthesized attributes of an open node are undefined and are thus associated with variables.

DEFINITION 2.7 (CONFIGURATION). A **configuration** Γ of an autonomous guarded attribute grammar is an S -sorted set of nodes $X \in \text{nodes}(\Gamma)$ each of which is associated with a defining equation in one of the following form where $\text{var}(\Gamma)$ is a set of variables associated with Γ :

Closed node: $X = R(X_1, \dots, X_k)$ where $X :: s$, and $X_i :: s_i$ for $1 \leq i \leq k$, and $\mathcal{U}(R) : s \rightarrow s_1 \dots s_k$ is the underlying production of rule R . Rule R is the **label** of node X and nodes X_1 to X_n are its **successor nodes**.

Open node: $X = s(t_1, \dots, t_n)\langle x_1, \dots, x_m \rangle$ where X is of sort s and t_1, \dots, t_k are terms with variables in $\text{var}(\Gamma)$ that represent the values of the inherited attributes of X , and x_1, \dots, x_m are variables in $\text{var}(\Gamma)$ associated with its synthesized attributes.

Each variable in $\text{var}(\Gamma)$ occurs at most once in a synthesized position. Otherwise stated $!\Gamma = \{!F \mid F \in \Gamma\}$ is an admissible labelling of the set of forms occurring in Γ . A node is called a **root node** when its sort is an axiom. Each node is the successor of a unique node, called its **predecessor**, except for the root nodes that are the successor of no other nodes. Hence a configuration is a set of trees – abstract-syntax trees of the underlying grammar – which we call the **artifacts** of the configuration. Each axiom is associated with a **map** made of the artifacts of the corresponding sort. A map thus collects the artifacts corresponding to a specific service of the GAG.

In order to specify the effect of applying a rule at a given node of a configuration (Definition 2.11) we first recall some notions about substitutions.

RECALL 2.8 (ON SUBSTITUTIONS). We identify a substitution σ on a set of variables $\{x_1, \dots, x_k\}$, called the **domain** of σ , with a system of equations

$$\{x_i = \sigma(x_i) \mid 1 \leq i \leq k\}$$

The set $\text{var}(\sigma) = \bigcup_{1 \leq i \leq k} \text{var}(\sigma(x_i))$ of variables of σ , is disjoint from the domain $\text{dom}(\sigma)$ of σ . Conversely a system of equations $\{x_i = t_i \mid 1 \leq i \leq k\}$ defines a substitution σ with $\sigma(x_i) = t_i$ if it is in **solved form**, i.e., none of the variables x_i appears in some of the terms t_j . In order to transform a system of equations $E = \{x_i = t_i \mid 1 \leq i \leq k\}$ into an equivalent system $\{x_i = t'_j \mid 1 \leq j \leq m\}$ in solved form one can iteratively replace an occurrence of a variable x_i in one of the right-hand side term t_j by its definition t_i until no variable x_i occurs in some t_j . This process terminates

when the relation $x_i \succ x_j \Leftrightarrow x_j \in \text{var}(\sigma(x_i))$ is acyclic. One can easily verify that, under this assumption, the resulting system of equation $SF(E) = \{x_i = t'_i \mid 1 \leq i \leq n\}$ in solved form does not depend on the order in which the variables x_i have been eliminated from the right-hand sides. When the above condition is met we say that the set of equations is **acyclic** and that it **defines** the substitution associated with the solved form. *End of Recall 2.8*

The composition of two substitutions σ, σ' , where $\text{var}(\sigma') \cap \text{dom}(\sigma) = \emptyset$, is denoted by $\sigma\sigma'$ and defined by $\sigma\sigma' = \{x = t\sigma' \mid x = t \in \sigma\}$. Similarly, we let $\Gamma\sigma$ denote the configuration obtained from Γ by replacing the defining equation $X = F$ of each open node X by $X = F\sigma$.

We now define more precisely when a rule is enabled at a given open node of a configuration and the effect of applying the rule. First, note that variables of a rule are formal parameters whose scope is limited to the rule. They can injectively be renamed in order to avoid clashes with variables names appearing in the configuration. Therefore we always assume that the set of variables of a rule R is disjoint from the set of variables of configuration Γ when applying rule R at a node of Γ . As informally stated in the previous section, a rule R applies at an open node X when its left-hand side $s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ matches with the definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$, namely the task attached to X in Γ .

First, the patterns p_i should match with the data d_i according to the usual pattern matching given by the following inductive statements

match $(c(p'_1, \dots, p'_k), c'(d'_1, \dots, d'_{k'}))$ with $c \neq c'$ fails
match $(c(p'_1, \dots, p'_k), c(d'_1, \dots, d'_k)) = \sum_{i=1}^k \text{match}(p'_i, d'_i)$
match $(x, d) = \{x = d\}$

where the sum $\sigma = \sum_{i=1}^k \sigma_i$ of substitutions σ_i is defined and equal to $\bigcup_{i \in 1..k} \sigma_i$ when all substitutions σ_i are defined and associated with disjoint sets of variables. Note that since no variable occurs twice in the whole set of patterns p_i , the various substitutions **match** (p_i, d_i) , when defined, are indeed concerned with disjoint sets of variables. Note also that **match** $(c(), c()) = \emptyset$.

DEFINITION 2.9. A form $F = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ **matches** with a service call $F' = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ –of the same sort– when

1. the patterns p_i 's matches with the data d_i 's, defining a substitution $\sigma_{in} = \sum_{1 \leq i \leq n} \text{match}(p_i, d_i)$,
2. the set of equations $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ is acyclic and defines a substitution σ_{out} .

The resulting substitution $\sigma = \text{match}(F, F')$ is given by $\sigma = \sigma_{out} \cup \sigma_{in}\sigma_{out}$.

REMARK 2.10. In most cases variables y_j do not appear in expressions d_i . And when it is the case one has only to check that patterns p_i 's matches with data d_i 's –substitution σ_{in} is defined– because then $\sigma_{out} = \{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ since the latter is already in solved form. Moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ because variables y_j do not appear in expressions $\sigma_{in}(x_i)$. *End of Remark 2.10*

DEFINITION 2.11 (APPLYING A RULE). Let $R = F_0 \rightarrow F_1 \dots F_k$ be a rule, Γ be a configuration, and X be an open node with definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ in Γ . We assume that R and Γ are defined over disjoint sets of variables. We say that R is **enabled** in X and write $\Gamma[R/X]$, if the left-hand side of R matches with the definition of X . Then applying rule R at node X transforms configuration Γ into Γ' , denoted as $\Gamma[R/X]\Gamma'$, with Γ' defined as follows:

$$\begin{aligned} \Gamma' &= \{X = R(X_1, \dots, X_k)\} \\ &\cup \{X_1 = F_1\sigma, \dots, X_k = F_k\sigma\} \\ &\cup \{X' = F\sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \end{aligned}$$

where $\sigma = \mathbf{match}(F_0, X)$ and X_1, \dots, X_k are new nodes added to Γ' .

Thus the first effect of applying rule R to an open node X is that X becomes a closed node with label R and successor nodes X_1 to X_k . The latter are new nodes added to Γ' . They are associated respectively with the instances of the k forms in the right-hand side of R obtained by applying substitution σ to these forms. The definitions of the other nodes of Γ are updated using substitution σ —or equivalently σ_{out} . This update has no effect on the closed nodes because their defining equations in Γ contain no variable.

We conclude this section with two results justifying Definition 2.11. Namely, Prop. 2.12 states that if R is a rule enabled in a node X_0 of a configuration Γ with $\Gamma[R/X_0]\Gamma'$ then Γ' is a configuration: Applying R cannot create a variable with several input occurrences. And Prop. 2.15 shows that substitution $\sigma = \mathbf{match}(F_0, X)$ resulting from the matching of the left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ of a rule R with the definition $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of an open node X is the most general unifier of the set of equations $\{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\}$.

PROPOSITION 2.12. If rule R is enabled in an open node X_0 of a configuration Γ and $\Gamma[R/X_0]\Gamma'$ then Γ' is a configuration.

PROOF. Let $R = F_0 \rightarrow F_1 \dots F_k$ with left-hand side $F_0 = s(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ and $X_0 = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ be the defining equation of X_0 in Γ . Since the values of synthesized attributes in the forms F_1, \dots, F_k are variables (by Definition 2.3) and since these variables are unaffected by substitution σ_{in} the synthesized attribute in the resulting forms $F_j\sigma_{in}$ are variables. The substitutions σ_{in} and σ_{out} substitute terms to the variables x_1, \dots, x_k appearing to the patterns and to the variables y_1, \dots, y_m respectively. Since x_i appears in an input position in R , it can appear only in an output position in the forms $!(F_1), \dots, !(F_k)$ and thus any variable of the term $\sigma_{in}(x_i)$ will appear in an output position in $!(F_i\sigma_{in})$. Similarly, since y_i appears in an input position in the form $!(s(u_1, \dots, u_m)\langle y_1, \dots, y_m \rangle)$, it can only appear in an output position in $!(F)$ for the others forms F of Γ . Consequently any variable of the term $\sigma_{out}(y_i)$ will appear in an output position in $!(F\sigma_{out})$ for any equation $X = F$ in Γ with $X \neq X_0$. It follows that the application of a rule cannot produce new occurrences of a variable in an input position and thus there cannot exist two occurrences $x_i^?$ of a same variable x in Γ' . Q.E.D.

Thus applying an enabled rule defines a binary relation on configurations.

DEFINITION 2.13. A configuration Γ' is **directly accessible** from Γ , denoted by $\Gamma[\]\Gamma'$, whenever $\Gamma[R/X]\Gamma'$ for some rule R enabled in node X of configuration Γ . Furthermore, a configuration Γ' is **accessible** from configuration Γ when $\Gamma[*]\Gamma'$ where $[\ast]$ is the reflexive and transitive closure of relation $[\]$.

Recall that a substitution σ unifies a set of equations E if $t\sigma = t'\sigma$ for every equation $t = t'$ in E . A substitution σ is more general than a substitution σ' , in notation $\sigma \leq \sigma'$, if $\sigma' = \sigma\sigma''$ for some substitution σ'' . If a system of equations has a some unifier, then it has—up to a bijective renaming of the variables in σ —a *most general unifier*. In particular a set of equations of the form $\{x_i = t_i \mid 1 \leq i \leq n\}$ has a unifier if and only if it is acyclic. In this case, the corresponding solved form is its most general unifier.

RECALL 2.14 (ON UNIFICATION). We consider sets $E = E_? \uplus E_ =$ containing equations of two kinds. An equation in $E_?$, denoted as $t \stackrel{?}{=} u$, represents a *unification goal* whose solution is a substitution σ such that $t\sigma = u\sigma$ —substitution σ unifies terms t and u . $E_ =$ contains equations of the form $x = t$ where variable x occurs only there, i.e., we do not have two equations with the same variable in their left-hand side and such a variable cannot either occur in any right-hand side of an equation in $E_ =$. A *solution* to E is any substitution σ whose domain is the set of variables occurring in the right-hand sides of equations in $E_ =$ such that the compound substitution made of σ and the set of equations $\{x = t\sigma \mid x = t \in E_ =\}$ unifies terms t and u for any equation $t \stackrel{?}{=} u$ in $E_?$. Two systems of equations are said to be *equivalent* when they have the same solutions. A *unification problem* is a set of such equations with $E_ = = \emptyset$, i.e., it is a set of unification goals. On the contrary E is said to be in *solved form* if $E_? = \emptyset$, thus E defines a substitution which, by definition, is the most general solution to E . Solving a unification problem E consists in finding an equivalent system of equations E' in solved form. In that case E' is a *most general unifier* for E .

Martelli and Montanari Unification algorithm [30] proceeds as follows. We pick up non deterministically one equation in $E_?$ and depending on its shape apply the corresponding transformation:

1. $c(t_1, \dots, t_n) \stackrel{?}{=} c(u_1, \dots, u_n)$: replace it by equations $t_1 \stackrel{?}{=} u_1, \dots, t_n \stackrel{?}{=} u_n$.
2. $c(t_1, \dots, t_n) \stackrel{?}{=} c'(u_1, \dots, u_m)$ with $c \neq c'$: halt with failure.
3. $x \stackrel{?}{=} x$: delete this equation.
4. $t \stackrel{?}{=} x$ where t is not a variable: replace this equation by $x \stackrel{?}{=} t$.
5. $x \stackrel{?}{=} t$ where $x \notin \text{var}(t)$: replace this equation by $x = t$ and substitute x by t in all other equations of E .
6. $x \stackrel{?}{=} t$ where $x \in \text{var}(t)$ and $x \neq t$: halt with failure.

The condition in (5) is the occur check. Thus the computation fails either if the two terms of an equation cannot be unified because their main constructors are different or because a potential solution of an equation is necessarily an infinite tree due to a recursive statement detected by the occur check. System E' obtained from E by applying one of these rules, denoted as $E \Rightarrow E'$, is clearly equivalent to E . We iterate this transformation as long as we do not encounter a failure and some equation remains in E . It can be proved that all these computations terminate and either the original unification problem E has a solution –a unifier– and every computation terminates –and henceforth produces a solved set equivalent to E describing a most general unifier of E – or E has no unifier and every computation fails. We let

$$\sigma = \mathbf{mgu}(\{t_i = u_i\}_{1 \leq i \leq n}) \text{ iff } \left\{ t_i \stackrel{?}{=} u_i \right\}_{1 \leq i \leq n} \Rightarrow^* \sigma$$

End of Recall 2.14

Note that (5) and (6) are the only rules that can be applied to solve a unification problem of the form $\{y_i \stackrel{?}{=} u_i \mid 1 \leq i \leq n\}$, where the y_i are distinct variables. The most general unifier exists when the occur check always holds, i.e., rule (5) always applies. The computation amounts to iteratively replacing an occurrence of a variable y_i in one of the right-hand side term u_j by its definition u_i until no variable y_i occurs in some u_j , i.e., (see Recall 2.8) when this system of equation is acyclic. Hence any acyclic set of equations $\{y_i = u_i \mid 1 \leq i \leq n\}$ defines the substitution $\sigma = \mathbf{mgu}(\{y_i = u_i \mid 1 \leq i \leq n\})$.

PROPOSITION 2.15. *If $F_0 = s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$, left-hand side of a rule R , matches with the definition $X = s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of an open node X then substitution $\sigma = \mathbf{match}(F_0, X)$ is the most general unifier of the set of equations $\{p_i = d_i \mid 1 \leq i \leq n\} \cup \{y_j = u_j \mid 1 \leq j \leq m\}$.*

PROOF. If a rule R of left-hand side $s_0(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle$ is triggered in node $X_0 = s_0(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ then by Def. 2.11 $\left\{ p_i \stackrel{?}{=} d_i \right\}_{1 \leq i \leq n} \cup \left\{ y_j \stackrel{?}{=} u_j \right\}_{1 \leq j \leq m} \Rightarrow^* \sigma_{in} \cup \left\{ y_j \stackrel{?}{=} u_j \sigma_{in} \right\}_{1 \leq j \leq m}$ using only the rules (1) and (5). Now, by applying iteratively rule (5) one obtains

$$\sigma_{in} \cup \left\{ y_j \stackrel{?}{=} u_j \sigma_{in} \right\}_{1 \leq j \leq m} \Rightarrow^* \sigma_{in} \cup \mathbf{mgu} \{y_j = u_j \sigma_{in}\}_{1 \leq j \leq m}$$

when the set of equations $\{y_j = u_j \sigma_{in}\}_{1 \leq j \leq m}$ satisfies the occur check. Then $\sigma_{in} + \sigma_{out} \Rightarrow^* \sigma$ again by using rule (5). *Q.E.D.*

REMARK 2.16. The converse of Prop. 2.15 does not hold. Namely, one shall not deduce from Proposition 2.15 that the relation $\Gamma[R/X_0]\Gamma'$ is defined whenever the left-hand side $\text{lhs}(R)$ of R can be unified with the definition $\text{def}(X_0, \Gamma)$ of X_0 in Γ with

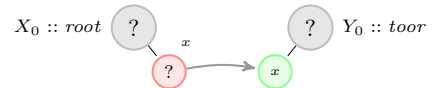
$$\begin{aligned} \Gamma' &= \{X_0 = R(X_1, \dots, X_k)\} \\ &\cup \{X_1 = F_1\sigma, \dots, X_k = F_k\sigma\} \\ &\cup \{X = F\sigma \mid (X = F) \in \Gamma \wedge X \neq X_0\} \end{aligned}$$

where $\sigma = \mathbf{mgu}(\text{lhs}(R), \text{def}(X_0, \Gamma))$, and X_1, \dots, X_k are new nodes added to Γ' . Indeed, when unifying $\text{lhs}(R)$ with $\text{def}(X_0, \Gamma)$ one may generate an equation of the form $x = t$ where x is a variable in an inherited data d_i and t is an instance of a corresponding subterm in the associated pattern p_i . This would correspond to a situation where information is sent to the context of a node through one of its inherited attribute! Stated differently, with this alternative definition some parts of the pattern p_i could actually be used to filter out the incoming data value d_i while some other parts of the same pattern would be used to transfer synthesized information to the context. *End of Remark 2.16*

2.4 Some Examples

In this section we illustrate the behaviour of guarded attribute grammars with two examples. Example 2.17 describes an execution of the attribute grammar of Example 2.2. The specification in Example 2.2 is actually an ordinary attribute grammar because the inherited attributes in the left-hand sides of rules are plain variables. This example shows how data are lazily produced and send in push mode through attributes. It also illustrates the role of the data links and their dynamic evolutions. Example 2.18 illustrates the role of the guards by describing two processes acting as coroutines. The first process sends forth a list of values to the second process and it waits for an acknowledgement for each message before sending the next one.

EXAMPLE 2.17 (EXAMPLE 2.2 CONTINUED). Let us consider the attribute grammar of Example 2.2 and the initial configuration $\Gamma_0 = \{X_0 = \text{root}()\langle x \rangle, Y_0 = \text{toor}()\langle \rangle\}$ shown next



The annotated version $!\Gamma_0 = \{!F \mid F \in \Gamma_0\}$ of configuration Γ_0 is

$$!\Gamma_0 = \left\{ X_0 = \text{root}()\langle x^? \rangle, Y_0 = \text{toor}()\langle \rangle \right\}$$

The data link from $x^?$ to $x^!$ says that the list of the leaves of the tree –that will stem from node X_0 – to be synthesized at node X_0 should be forwarded to the inherited attribute of Y_0 .

This tree is not defined in the initial configuration Γ_0 . One can start developing it by applying rule $\text{Root} : \text{root}()\langle u \rangle \rightarrow \text{bin}(\text{Nil})\langle u \rangle$ at node $X_0 :: \text{root}$. Actually the left-hand side $\text{root}()\langle u \rangle$ of rule Root matches with the definition $\text{root}()\langle x \rangle$ of X_0 with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = u\}$. Thus $\Gamma_0[\text{Root}/X_0]\Gamma_1$ where the annotated configuration $!\Gamma_1$ is given in Figure 4.

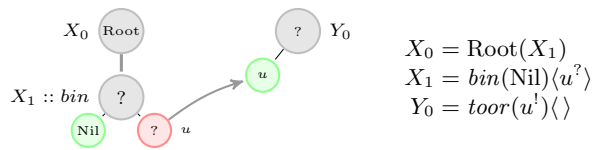


Figure 4: Configuration Γ_1

Note that substitution $\sigma_{out} = \{x = u\}$ replaces the data link $(x^?, x^!)$ by a new link $(u^?, u^!)$ with the same target and whose source has been moved from the synthesized attribute of X_0 to the synthesized attribute of X_1 .



The tree may be refined by applying rule

$$\text{Fork} : bin(x)\langle y \rangle \rightarrow bin(z)\langle y \rangle bin(x)\langle z \rangle$$

at node $X_1 :: bin$ since its left-hand side $bin(x)\langle y \rangle$ matches with the definition $bin(Nil)\langle u \rangle$ of X_1 with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{u = y\}$. Hence $\Gamma_1[\text{Fork}/X_1]\Gamma_2$ where $!\Gamma_2$ is given in Figure 5.

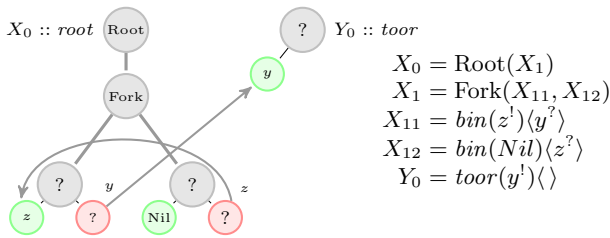


Figure 5: Configuration Γ_2

Rule $\text{Leaf}_c : bin(x)\langle \text{Cons}_c(x) \rangle \rightarrow$ applies at node X_{12} since its left-hand side $bin(x)\langle \text{Cons}_c(x) \rangle$ matches with the definition $bin(Nil)\langle z \rangle$ of X_{12} with $\sigma_{in} = \{x = Nil\}$ and $\sigma_{out} = \{z = \text{Cons}_c(Nil)\}$. Hence $\Gamma_2[\text{Leaf}_c/X_{12}]\Gamma_3$ where the annotated configuration $!\Gamma_3$ is given in Figure 6.

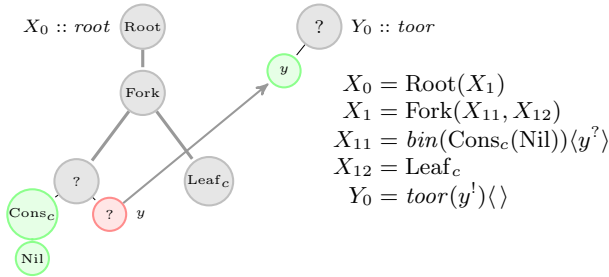


Figure 6: Configuration Γ_3

As a result of substitution $\sigma_{out} = \{z = \text{Cons}_c(Nil)\}$ the value $\text{Cons}_c(Nil)$ is transmitted through the link $(z^?, z^!)$ and this link disappears.

Rule $\text{Fork} : bin(x)\langle u \rangle \rightarrow bin(z)\langle u \rangle bin(x)\langle z \rangle$ may apply at node X_{11} : its left-hand side $bin(x)\langle u \rangle$ matches with the definition $bin(\text{Cons}_c(Nil))\langle y \rangle$ of X_{11} with $\sigma_{in} = \{x = \text{Cons}_c(Nil)\}$ and $\sigma_{out} = \{y = u\}$. Hence $\Gamma_3[\text{Fork}/X_{11}]\Gamma_4$ with configuration $!\Gamma_4$ given in Figure 7.

Rule $\text{Leaf}_a : bin(x)\langle \text{Cons}_a(x) \rangle \rightarrow$ applies at node X_{111} since its left-hand side $bin(x)\langle \text{Cons}_a(x) \rangle$ matches with the definition $bin(z)\langle u \rangle$ of X_{111} with $\sigma_{in} = \{x = z\}$ and $\sigma_{out} =$

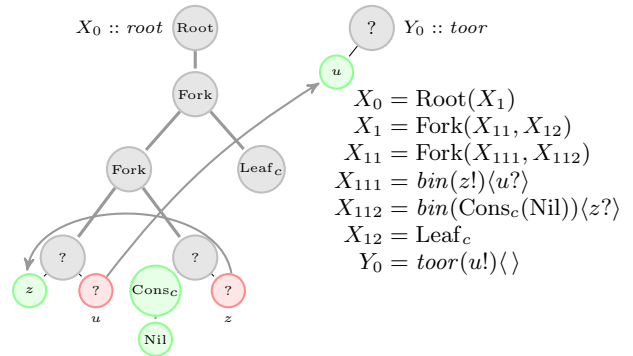


Figure 7: Configuration Γ_4

$\{u = \text{Cons}_a(z)\}$. Hence $\Gamma_4[\text{Leaf}_a/X_{111}]\Gamma_5$ with configuration $!\Gamma_5$ given in Figure 8.

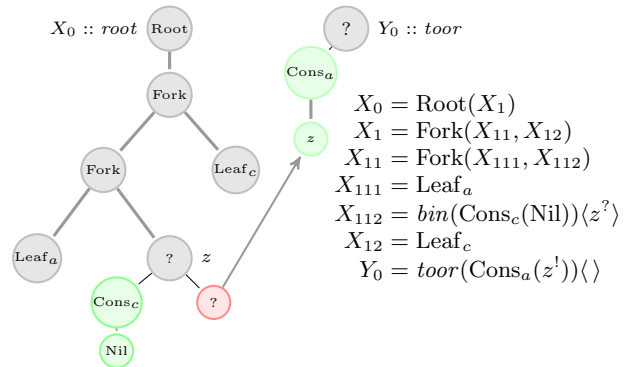


Figure 8: Configuration Γ_5

Using substitution $\sigma_{out} = \{u = \text{Cons}_a(z)\}$ the data $\text{Cons}_a(z)$ is transmitted through the link $(u^?, u^!)$ which, as a result, disappears. A new link $(z^?, z^!)$ is created so that the rest of the list, to be synthesized in node X_{112} can later be forwarded to the inherited attribute of Y_0 .

Finally one can apply rule $\text{Leaf}_b : bin(x)\langle \text{Cons}_a(x) \rangle \rightarrow$ at node X_{112} since its left-hand side matches with the definition $bin(\text{Cons}_c(Nil))\langle z \rangle$ of X_{112} with $\sigma_{in} = \{x = \text{Cons}_c(Nil)\}$ and $\sigma_{out} = \{z = \text{Cons}_b(\text{Cons}_c(Nil))\}$.

Therefore $\Gamma_5[\text{Leaf}_b/X_{112}]\Gamma_6$ with configuration $!\Gamma_6$ given in Figure 9.

Now the tree rooted at node X_0 is closed –and thus it no longer holds attributes– and the list of its leaves has been entirely forwarded to the inherited attribute of node Y_0 . Note that the recipient node Y_0 could have been refined in parallel with the changes of configurations just described.

End of Exple 2.17

The above example shows that data links are used to transmit data in push mode from a source vertex v –the input occurrence $x^?$ of a variable x – to some target vertex v' – an output occurrence $x^!$ of the same variable. These links

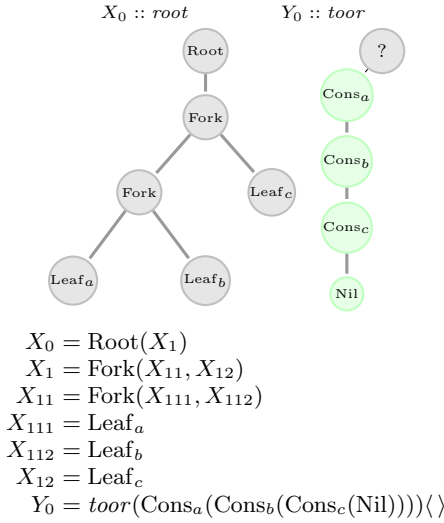


Figure 9: Configuration Γ_6

(x^1, x^2) are *transient* in the sense that they disappear as soon as variable x gets defined by the substitution σ_{out} induced by the application of a rule in some open node of the current configuration. If $\sigma_{out}(x)$ is a term t , not reduced to a variable, with variables x_1, \dots, x_k then vertex v' is refined by the term $t[x_i^1/x_i]$ and new vertices v'_i —associated with these new occurrences of x_i in an output position— are created. The original data link (x^2, x^1) is replaced by all the corresponding instances of (x_i^2, x_i^1) . Consequently, a target is replaced by new targets which are the recipients for the subsequent pieces of information —maybe none because no new links are created when t contains no variable. If the term t is a variable y then the link (x^2, x^1) is replaced by the link (y^2, y^1) with the same target and whose source, the (unique) occurrence x^2 of variable x , is replaced by the (unique) occurrence y^2 of variable y . Therefore the direction of the flow of information is in both cases preserved: Channels can be viewed as “generalized streams” —that can fork or vanish— through which information is pushed incrementally.

EXAMPLE 2.18. Figure 10 shows a guarded attribute grammar that represents two coroutines communicating through lazy streams. Each process alternatively sends and receives data. More precisely the second process sends an acknowledgment —message $?b$ — upon reception of a message sent by the left process. Initially or after reception of an acknowledgment of its previous message the left process can either send a new message or terminate the communication.

Production $!a : q_1(x')\langle a(y') \rangle \leftarrow q_2(x')\langle y' \rangle$ applies at node X_1 of configuration

$$\Gamma_1 = \{ X = X_1 \| X_2, X_1 = q_1(x)\langle y \rangle, X_2 = q_2(y)\langle x \rangle \}$$

shown in Figure 11 because its left-hand side $q_1(x')\langle a(y') \rangle$ matches with the definition $q_1(x)\langle y \rangle$ of X_1 with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = a(y')\}$. We get configuration

$$\Gamma_2 = \left\{ \begin{array}{l} X = X_1 \| X_2, X_1 = !a(X_{11}), \\ X_2 = q_2'(a(y'))\langle x \rangle, X_{11} = q_2(x)\langle y' \rangle \end{array} \right\}$$

shown on the middle of Figure 11.

Production $?a : q_2'(a(y))\langle x' \rangle \leftarrow q_1(y)\langle x' \rangle$ applies at node X_2 of Γ_2 because its left-hand side $q_2'(a(y))\langle x' \rangle$ matches with the definition $q_2'(a(y'))\langle x \rangle$ of X_2 with $\sigma_{in} = \{y = y'\}$ and $\sigma_{out} = \{x = x'\}$. We get configuration

$$\Gamma_3 = \left\{ \begin{array}{l} X = X_1 \| X_2, X_1 = !a(X_{11}), X_2 = ?a(X_{21}), \\ X_{11} = q_2(x')\langle y' \rangle, X_{21} = q_1'(y')\langle x' \rangle \end{array} \right\}$$

shown on the right of Figure 11.

The corresponding acknowledgment may be sent and received leading to configuration

$$\Gamma_5 = \Gamma \cup \{ X_{111} = q_1(x)\langle y \rangle, X_{211} = q_2'(y)\langle x \rangle \}.$$

$$\text{where } \Gamma = \left\{ \begin{array}{l} X = X_1 \| X_2, X_1 = !a(X_{11}), X_2 = ?a(X_{21}), \\ X_{21} = !b(X_{211}), X_{11} = ?b(X_{111}) \end{array} \right\}.$$

The process on the left may decide to end communication by applying production $!stop : q_1(x')\langle stop \rangle \leftarrow$ at X_{111} with $\sigma_{in} = \{x' = x\}$ and $\sigma_{out} = \{y = stop\}$ leading to configuration

$$\Gamma_6 = \Gamma \cup \{ X_{111} = !stop, X_{211} = q_2'(stop)\langle x \rangle \}.$$

The reception of this message by the process on the right corresponds to applying production $?stop : q_2(stop)\langle y \rangle \leftarrow$ at X_{211} with $\sigma_{in} = \emptyset$ and $\sigma_{out} = \{x = y\}$ leading to configuration

$$\Gamma_7 = \Gamma \cup \{ X_{111} = !stop, X_{211} = ?stop \}.$$

Note that variable x appears in an input position in Γ_6 and has no corresponding output occurrence. This means that the value of x is not used in the configuration. When production $?stop$ is applied in node X_{211} variable y is substituted to x . Variable y has an output occurrence in production $?stop$ and no input occurrence meaning that the corresponding output attribute is not defined by the semantic rules. As a consequence this variable simply disappears in the resulting configuration Γ_7 . If variable x was used in Γ_6 then the output occurrences of x would have been replaced by (output occurrences) of variable y that will remain undefined —no value will be substituted to y in subsequent transformations— until these occurrences of variables may possibly disappear.

End of Exple 2.18

3. COMPOSITION OF GAG

In this section we define the behavior of potentially non-autonomous guarded attributed grammars to account for systems that call for external services: A guarded attribute grammar providing some set of services may contain terminal symbols, namely symbols that do not occur in the left-hand sides of rules. These terminal symbols are interpreted as calls to external services that are associated with some other guarded attribute grammar. We introduce a composition of guarded attribute grammars and show that the behavior of the composite guarded attribute grammar can be recovered from the behavior of its components.

Recall that the behavior of an active workspace is given by a guarded attribute grammar G . Its configuration is a set of

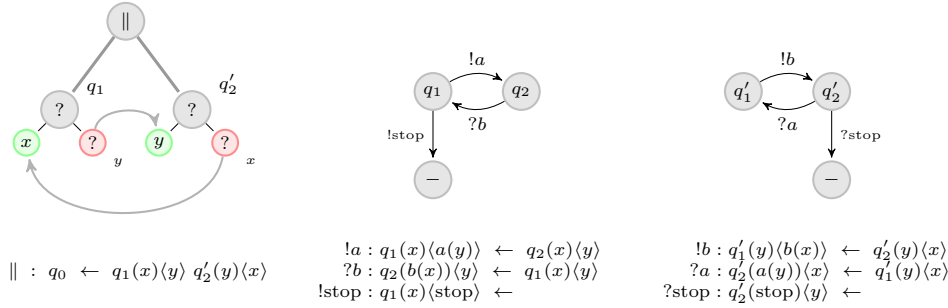


Figure 10: Coroutines with lazy streams

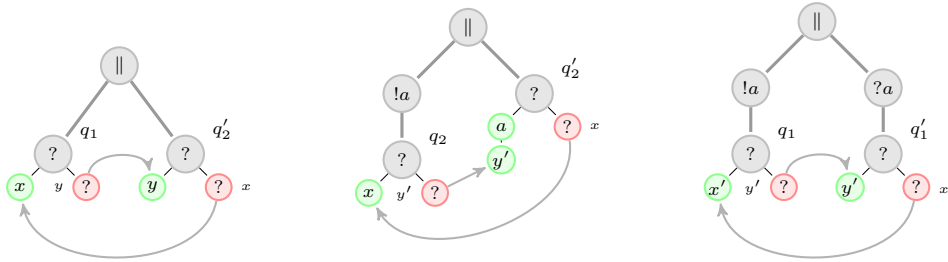


Figure 11: $\Gamma_1[!a/X_1]\Gamma_2[?a/X_2]\Gamma_3$

maps associated with each of the axioms, or services, of the grammar. A map contains the artifacts generated by calls to the corresponding service. We assume that each active workspace has a namespace $ns(G)$ used for the nodes X of its configuration, the variables x occurring in the values of attributes of these nodes, and also for references to variables belonging to others active workspaces –its subscriptions. Hence we have a name generator that produces unique identifiers for each newly created variable of the configuration. Furthermore, we assume that the name of a variable determines its *location*, namely the active workspace it belongs to. A configuration is given by a set of equations as stated in Definition 2.7 with the following changes.

1. A node associated with a terminal symbol is associated with no equation –it corresponds a service call that initiates an artifact in another active workspace.
2. We have equations of the form $y = x$ stating that distant variable y subscribes to the value of local variable x .
3. We have equations of the form $Y = X$ where Y is the distant node that created the artifact rooted at local node X . Hence X is a root node.

Futhermore we add an input and an output buffers to each configuration Γ . They contains messages respectively received from and send to distant locations. A message in the input buffer $in(\Gamma)$ is one of the following types.

1. $Y = s(t_1, \dots, t_n)(y_1, \dots, y_m)$ tells that distant node Y calls service $s \in \mathbf{axioms}(G)$. When reading this message we create a new root node X –the root of

the artifact associated with the service call. And values t_1, \dots, t_n are assigned to the inherited attributes of node X while the distant variables y_1, \dots, y_m subscribe to the values of its synthesized attributes. We replace variable Y by a dummy variable (wildcard: $_$) when this service call is not invoked from a distant active workspace but from an external user of the system.

2. $x = t$ tells that local variable x receives the value t from a subscription created at a distant location.
3. $y = x$ states that distant variable y subscribes to the value of local variable x .

Symmetrically, a message in the output buffer $out(\Gamma)$ is one of the following types.

1. $X = s(t_1, \dots, t_n)(y_1, \dots, y_m)$ tells that local node X calls the external service s –a terminal symbol– with values t_1, \dots, t_n assigned to the inherited attributes. And the local variables y_1, \dots, y_m subscribe to the values of the synthesized attributes of the distant node where the artifact generated by this service call will be rooted at.
2. $y = t$ tells that value t is sent to distant variable y according to a subscription made for this variable.
3. $x = y$ states that local variable x subscribes to the value of distant variable y .

The behavior of a guarded attribute grammar is given by relation $\Gamma \xrightarrow[e]{M} \Gamma'$ stating that event e transforms configuration Γ into Γ' and adds the set of messages M to the

output buffer. An event is the application of a rule R to a node X of configuration Γ or the consumption of a message from its input buffer. Let us start with the former kind of event: $e = R/X$. let $X = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle \in \Gamma$ and $R = F \rightarrow F_1 \cdots F_k$ be a rule whose left-hand side matches with X producing substitution $\sigma = \mathbf{match}(F, X)$. All variables occurring in the definition of node X are local. We recall that in order to avoid name clashes we rename all the variables of rule R with fresh names. We use the local name generator for that purpose. Hence all variables of R are also local variables –freshly created ones. Therefore all variables used and defined by substitution σ are local variables. Then we let $\Gamma \xrightarrow[M]{R/X} \Gamma'$ where

$$\begin{aligned} \Gamma' &= \{X = R(X_1, \dots, X_k)\} \\ &\cup \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in N\} \\ &\cup \{X' = F\sigma \mid (X' = F) \in \Gamma \wedge X' \neq X\} \\ &\cup \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ is a variable}\} \\ &\cup \{Y = X \mid (Y = X) \in \Gamma\} \\ M &= \{X_i = F_i\sigma \mid X_i :: s_i \text{ and } s_i \in T\} \\ &\cup \{y = y_j\sigma \mid (y = y_j) \in \Gamma \text{ and } y_j\sigma \text{ not a variable}\} \end{aligned}$$

where X_1, \dots, X_k are new names in $ns(G)$. Note that when a distant variable y subscribes to some synthesized attribute of node X , namely $(y = y_i) \in \Gamma$, two situations can occur depending on whether $y_j\sigma$ is a variable or not. When $y_j\sigma = x$ is a (local) variable the subscription $y = y_i$ is replaced by subscription $y = x$: Variable y_i delegates the production of the required value to x . This operation is totally transparent to the location that initiated the subscription. But as soon as some value is produced – $y_j\sigma$ is not a variable – it is immediately sent to the subscribing variable even when this value contains variables: Values are produced and send incrementally.

Let us now consider the event associated with the consumption of a message $m \in out(\Gamma)$ in the output buffer.

1. If $m = (Y = s(t_1, \dots, t_n)\langle y_1, \dots, y_q \rangle)$ then

$$\begin{aligned} \Gamma' &= \Gamma \cup \{\bar{Y} = s(\bar{t}_1, \dots, \bar{t}_n)\langle \bar{y}_1, \dots, \bar{y}_q \rangle\} \\ &\cup \{y_j = \bar{y}_j \mid 1 \leq j \leq q\} \cup \{Y = \bar{Y}\} \end{aligned}$$

where \bar{Y} , the variables \bar{x} for $x \in var(t_i)$ and the variables \bar{y}_j are new names in $ns(G)$, $\bar{t} = t[\bar{x}/x]$, and $M = \{\bar{x} = x \mid x \in var(t_i) \ 1 \leq i \leq n\}$.

2. If $m = (x=t)$ then $\Gamma' = \Gamma[x = t[\bar{y}/y]]$ where \bar{y} are new names in $ns(G)$ associated with the variables y in t and $M = \{\bar{y} = y \mid y \in var(t)\}$.
3. If $m = (y=x)$ then $\Gamma' = \Gamma \cup \{y = x\}$ and $M = \emptyset$.

We know define the guarded attribute grammar resulting from the composition of a set of smaller guarded attribute grammars.

DEFINITION 3.1 (COMPOSITION OF GAG).

Let G_1, \dots, G_p be guarded attribute grammars with disjoint sets of non terminal symbols such that each terminal symbol of a grammar G_i that belongs to another grammar G_j must be an axiom of the latter: $T_i \cap S_j = T_i \cap \mathbf{axioms}(G_j)$ where $s \in T_i \cap \mathbf{axioms}(G_j)$ means that grammar G_i uses service

s of G_j . Their **composition**, denoted as $G = G_1 \oplus \dots \oplus G_p$, is the guarded attribute grammar whose set of rules is the union of the rules of the G_i s and with set of axioms $\mathbf{axioms}(G) = \cup_{1 \leq i \leq p} \mathbf{axioms}(G_i)$. We say that the G_i are the **local grammars** and G the **global grammar** of the composition. If some axiom of the resulting global grammar calls itself recursively we apply the transformation described in Rem. 2.4.

One may also combine this composition with a restriction operator, $G \downarrow_{\mathbf{ax}}$, if the global grammar offers only a subset $\mathbf{ax} \subseteq \cup_{1 \leq i \leq p} \mathbf{axioms}(G_i)$ of the services provided by the local grammars.

Note that the set of terminal symbols of the composition is given by

$$T = (\cup_{1 \leq i \leq p} T_i) \setminus (\cup_{1 \leq i \leq p} \mathbf{axioms}(G_i))$$

i.e., its set of external services are all external services of a local grammar but those which are provided by some other local grammar. Note also that the set of non-terminal symbols of the global grammar is the (disjoint) union of the set of non-terminal symbols of the local grammars: $N = \cup_{1 \leq i \leq p} N_i$. This partition can be used to retrieve the local grammar by taking the rules of the global grammar whose sorts in their left-hand side belongs to the given equivalent class. Of course not every partition of the set of non-terminal symbols of a guarded attribute grammar corresponds to a decomposition into local grammars. To decompose a guarded attribute grammar into several components one can proceed as follows:

1. Select a partition $\mathbf{axiom}(G) \subseteq \cup_{1 \leq i \leq n} \mathbf{axioms}_i$ of the set of axioms. These sets are intended to represent the services associated with each of the local grammars.
2. Construct the local grammar G_i associated with services \mathbf{axioms}_i by first taking the rules whose left-hand sides are forms of sort $s \in \mathbf{axioms}_i$, and then iteratively adding to G_i the rules whose left-hand sides are forms of sort $s \in N \setminus \cup_{j \neq i} \mathbf{axioms}_j$ such that s appears in the right-hand side of a rule previously added to G_i .
3. If appropriate, namely when a same rule is copied in several components, rename the non-terminal symbols of the local grammars to ensure that they have disjoint sets of non-terminal symbols.

The above transformation can duplicate rules in G in the resulting composition $\bar{G} = G_1 \oplus \dots \oplus G_n$ but does not radically change the original specification.

Configurations of guarded attribute grammars are enriched with subscriptions –equations of the form $y = x$ – to enable communication between the various active workspaces. One might dispense with equations of the form $Y = X$ in the operational semantics of guarded attribute grammars. But they facilitate the proof of correctness of this composition (Prop. 3.2) by easing the reconstruction of the global configuration from its set of local configurations. Indeed, the global configuration can be recovered as $\Gamma = \Gamma_1 \oplus \dots \oplus \Gamma_p$ where operator \oplus consists in taking the union of the systems of equations given as arguments and simplifying the

resulting system by elimination of the copy rules: We drop each equation of the form $Y = X$ (respectively $y = x$) and replace each occurrence of Y by X (resp. of y by x).

Let $G = G_1 \oplus \dots \oplus G_p$ be a composition, Γ_i be a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \dots \oplus \Gamma_p$ the corresponding global configuration. Since the location of a variable derives from its identifier we know the location of destination of every message in the output buffer of a component. If M is a set of messages we let $M_i \subseteq M$ denote the set of messages in M to be forwarded to G_i . Their union $M_i = \cup_{1 \leq i \leq p} M_i$ is the set of *internal* messages that circulate between the local grammars. The rest $M_G = M \setminus M_i$ is the set of messages that remain in the output buffer of the global grammar –the *global* messages.

The join dynamics of the local grammars can be derived as follows from their individual behaviors, where e stands for R/X or a message m :

1. If $\Gamma_i \xrightarrow[M]{e} \Gamma'_i$ then $\Gamma \xrightarrow[M]{e} \Gamma'$ with $\Gamma_j = \Gamma'_j$ for $j \neq i$.
2. If $\Gamma \xrightarrow[M]{e} \Gamma'$ and $\Gamma' \xrightarrow[M']{m} \Gamma''$ for $m \in M$ then

$$\Gamma \xrightarrow[M \setminus \{m\} \cup M']{e} \Gamma''.$$

The correctness of the composition is given by the following proposition. It states that (i) every application of a rule can immediately be followed by the consumption of the local messages generated by it, and (ii) the behavior of the global grammar can be recovered from the joint behavior of its components where all internal messages are consumed immediately.

PROPOSITION 3.2. *Let $G = G_1 \oplus \dots \oplus G_p$ be a composition, Γ_i a configuration of G_i for $1 \leq i \leq p$ and $\Gamma = \Gamma_1 \oplus \dots \oplus \Gamma_p$ the corresponding global configuration. Then*

1. $\Gamma \xrightarrow[M]{R/X} \Gamma' \implies \exists! \Gamma''$ such that $\Gamma \xrightarrow[M_G]{R/X} \Gamma''$
2. $\Gamma \xrightarrow[M_G]{e} \Gamma' \iff \Gamma \xrightarrow[M_G]{e} \Gamma'$

PROOF. The first statement follows from the fact that relation $\Gamma \xrightarrow[M]{e} \Gamma'$ is deterministic (M and Γ' are uniquely defined from Γ and e) and the application of a rule produces, directly or indirectly, a finite number of messages.

- If m is of the form $Y = s(t_1, \dots, t_n)\langle y_1, \dots, y_n \rangle$, then consuming m results in adding new equations to the local configuration that receives it, and generating a set of messages of the form $\bar{x} = x$ that can hence be consumed by the location that will receive them without generating new messages (case 3 below).
- If $m = (x = t)$, then consumption of the message results in production of new variables, and a new (finite) set of messages of the form $\bar{y} = y$ that can then be consumed by the location that send m without producing any new message.
- If $m = (y = x)$ then consuming the message results in adding an equation $y = x$ to the local configuration without generating any new message.

The second statement follows from the fact that the construction of a global configuration $\Gamma = \Gamma_1 \oplus \dots \oplus \Gamma_p$ amounts to consuming all the local messages between the components. Q.E.D.

COROLLARY 3.3. *Let $G = G_1 \oplus \dots \oplus G_p$ be a composition where G is an autonomous guarded attribute grammar and Γ be a configuration of G . Then*

$$\Gamma[R/X]\Gamma' \iff \Gamma \xrightarrow[\emptyset]{R/X} \Gamma'$$

A configuration is *stable* when all internal messages are consumed. The behavior of the global grammar is thus given as the restriction of the join behavior of the components to the set of stable configurations. This amounts to imposing that every event of the global configuration is immediately followed by the consumptions of the internal messages generated by the event. However if the various active workspaces are distributed at distant locations on an asynchronous architecture, one can never guarantee that no internal message remains in transit, or that some message is received but not yet consumed in some distant location. In order to ensure a correct distribution we therefore need a monotony property stating that (i) a locally enabled rule cannot become disabled by the arrival of a message and (ii) local actions and incoming messages can be swapped. We identify in the next section a class of guarded attribute grammars having this monotony property and which thus guarantees a safe implementation of distributed active workspaces.

4. PROPERTIES OF GAG

In this section we investigate some formal properties of guarded attribute grammars. We first turn our attention to *input-enabled* guarded attribute grammars to guarantee that the application of a rule in an open node is a monotonous and confluent operation. This property is instrumental for the distribution of a GAG specification on an asynchronous architecture. We then consider *soundness*, a classical property of case management systems stating that any case introduced in the system can reach completion. We show that this property is undecidable. Nonetheless, soundness is preserved by hierarchical composition –a restrictive form of modular composition. This opens up the ability to obtain a large class of specifications that are sound by construction, if we start from basic specifications that are known to be sound by some ad-hoc arguments.

4.1 Distribution of a GAG

We say that rule R is **triggered** in node X if substitution σ_{in} –given in def. 2.9– is defined: Patterns p_i match the data d_i . As shown by the following example one can usually suspect a flaw in a specification when a triggered transition is not enabled due to the fact that the system of equations $\{y_j = u_j \sigma_{in} \mid 1 \leq j \leq m\}$ is cyclic.

EXAMPLE 4.1. Let us consider the guarded attribute grammar given by the following rules:

$$\begin{aligned} P &: s_0(\langle \rangle) \rightarrow s_1(a(x))\langle x \rangle \quad s_2(x)\langle \rangle \\ Q &: s_1(y)\langle a(y) \rangle \rightarrow \\ R &: s_2(a(z))\langle \rangle \rightarrow \end{aligned}$$

Applying P in node X_0 of configuration $\Gamma_0 = \{X_0 = s_0()\langle\rangle\}$ leads to configuration

$$\Gamma_1 = \{X_0 = P(X_1, X_2); X_1 = s_1(a(x))\langle x \rangle; X_2 = s_2(x)\langle\rangle\}$$

Rule Q is triggered in node X_1 with $\sigma_{in} = \{y = a(x)\}$ but the occur check fails because variable x occurs in $a(y)\sigma_{in} = a(a(x))$. Alternatively, we could drop the occur check and instead adapt the fixed point semantics for attribute evaluation defined in [7, 31] in order to cope with infinite data structures. More precisely, we let σ_{out} be defined as the least solution of system of equations $\{y_i = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ — assuming these equations are guarded, i.e., that there is no cycle of copy rules in the link graph of any accessible configuration. In that case the infinite tree a^ω is substituted to variable x and the unique maximal computation associated with the grammar is given by the infinite tree $P(Q, R^\omega)$. In Definition 2.11 we have chosen to restrict to finite data structures which seems a reasonable assumption in view of the nature of systems we want to model. The occur check is used to avoid recursive definitions of attribute values. The given example, whose most natural interpretation is given by fixed point computation, should in that respect be considered as ill-formed. And indeed this guarded attribute grammar is not *sound* — a notion presented in Section 4.2 — because configuration Γ_1 is not closed (it still contains open nodes) but yet it is a terminal configuration that enables no rule. Hence it represents a case that can not be terminated.

End of Exple 4.1

The fact that triggered rules are not enabled can also impact the distributability of a grammar as shown by the following example.

EXAMPLE 4.2. Let us consider the GAG with the following rules:

$$\begin{aligned} P: s()\langle\rangle &\rightarrow s_1(x)\langle y \rangle \ s_2(y)\langle x \rangle \\ Q: s_1(z)\langle a(z) \rangle &\rightarrow \\ R: s_2(u)\langle a(u) \rangle &\rightarrow \end{aligned}$$

Rule P is enabled in configuration $\Gamma_0 = \{X_0 = s()\langle\rangle\}$ with $\Gamma_0[P/X_0]\Gamma_1$ where

$$\Gamma_1 = \{X_0 = P(X_1, X_2); X_1 = s_1(x)\langle y \rangle, X_2 = s_2(y)\langle x \rangle\}.$$

In configuration Γ_1 rules Q and R are enabled in nodes X_1 and X_2 respectively with $\Gamma_1[Q/X_1]\Gamma_2$ where

$$\Gamma_2 = \{X_0 = P(X_1, X_2); X_1 = Q, X_2 = s_2(a(x))\langle x \rangle\}$$

and $\Gamma_1[R/X_2]\Gamma_3$ where

$$\Gamma_3 = \{X_0 = P(X_1, X_2); X_1 = s_2(a(y))\langle y \rangle, X_2 = R\}$$

Now rule R is triggered but not enabled in node X_2 of configuration Γ_2 because of the cyclicity of $\{x = a(a(x))\}$. Similarly, rule Q is triggered but not enabled in node X_3 of configuration Γ_3 . There is a conflict between the application of rules R and Q in configuration Γ_1 . When the grammar is distributed in such a way that X_1 and X_2 have distinct locations, the specification is not implementable.

End of Exple 4.2

We first tackle the problem of safe distribution of a GAG specification on an asynchronous architecture by limiting

ourselves to standalone systems. Hence to autonomous guarded attribute grammars. At the end of the section we show that this property can be verified in a modular fashion if the grammar is given as the composition of local (and thus non-autonomous) grammars.

DEFINITION 4.3 (ACCESSIBLE CONFIGURATIONS).

Let G be an autonomous guarded attribute grammar. A *case* $c = s(t_1, \dots, t_n)\langle x_1, \dots, x_m \rangle$ is a ground instantiation of service s , an axiom of the grammar, i.e., the values t_i of the inherited attributes are ground terms. It means that it is a service call which already contains all the information coming from the environment of the guarded attribute grammar. An **initial configuration** is any configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with a case c . An **accessible configuration** is any configuration accessible from an initial configuration.

Substitution σ_{in} , given by pattern matching, is monotonous w.r.t. incoming information and thus it causes no problem for a distributed implementation of a model. However substitution σ_{out} is not monotonous since it may become undefined when information coming from a distant location makes the match of output attributes a cyclic set of equations, as illustrated by example 4.2.

DEFINITION 4.4. An autonomous guarded attribute grammar is **input-enabled** if every rule that is triggered in an accessible configuration is also enabled.

If every form $s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ occurring in some reachable configuration is such that variables y_j do not appear in expressions d_i then by Remark 2.10 the guarded attribute grammar is input-enabled — moreover $\sigma = \sigma_{out} \cup \sigma_{in}$ for every enabled rule. This property is clearly satisfied for guarded L-attributed grammars which consequently constitute a class of input-enabled guarded attribute grammars.

DEFINITION 4.5 (L-ATTRIBUTED GRAMMARS).

A guarded attribute grammar is left-attributed, in short a LGAG, if any variable that is used in an inherited position in some form F of the right-hand side of a rule is either a variable defined in a pattern in the left-hand side of the rule or a variable occurring at a synthesized position in a form which appears at the left of F , i.e., inherited information flows from top-to-bottom and left-to-right between sibling nodes.

We call the *substitution induced by a sequence* $\Gamma[*]\Gamma'$ the corresponding composition of the various substitutions associated respectively with each of the individual steps in the sequence. If X is an open node in both Γ and Γ' , i.e., no rules are applied at node X in the sequence, then we get $X = s(d_1\sigma, \dots, d_n\sigma)\langle y_1, \dots, y_m \rangle \in \Gamma'$ where

$$X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle \in \Gamma$$

and σ is the substitution induced by the sequence.

PROPOSITION 4.6 (MONOTONY).

Let Γ be an accessible configuration of an input-enabled GAG, $X = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle \in \Gamma$ and σ be the substitution induced by some sequence starting from Γ . Then

$$\Gamma[P/X]\Gamma' \text{ implies } \Gamma\sigma[P/X]\Gamma'\sigma.$$

PROOF. Direct consequence of Definition 2.3 due to the fact that

1. $\text{match}(p, d\sigma) = \text{match}(p, d)\sigma$, and
2. $\text{mgu}(\{y_j = u_j\sigma \mid 1 \leq j \leq m\}) = \text{mgu}(\{y_j = u_j \mid 1 \leq j \leq m\})\sigma$.

The former is trivial and the latter follows by induction on the length of the computation of the most general unifier $\text{-relation} \Rightarrow^*$ using rule (5) only. Note that the assumption that the guarded attribute grammar is input-enabled is crucial because in the general case it could happen that the set $\{y_j = u_j\sigma_{in} \mid 1 \leq j \leq m\}$ satisfies the occur check whereas the set $\{y_j = u_j(\sigma_{in}\sigma) \mid 1 \leq j \leq m\}$ does not satisfy the occur check. Q.E.D.

Proposition 4.6 is instrumental for the distributed implementation of guarded attribute grammars. Namely it states that new information coming from a distant asynchronous location refining the value of some input occurrences of variables of an enabled rule do not prevent the rule to apply. Thus a rule that is locally enabled can freely be applied regardless of information that might further refine the current partial configuration. It means that conflicts arise only from the existence of two distinct rules enabled in the same open node. Hence the only form of non-determinism corresponds to the decision of a stakeholder to apply one particular rule among those enabled in a configuration. This is expressed by the following confluence property.

COROLLARY 4.7. *Let Γ be an accessible configuration of an input enabled GAG. If $\Gamma[P/X]\Gamma_1$ and $\Gamma[Q/Y]\Gamma_2$ with $X \neq Y$ then $\Gamma_2[P/X]\Gamma_3$ and $\Gamma_1[Q/Y]\Gamma_3$ for some configuration Γ_3 .*

Note that, by Corollary 4.7, the artifact contains a full history of the case in the sense that one can reconstruct from the artifact the complete sequence of applications of rules leading to the resolution of the case —up to the commutation of independent elements in the sequence.

REMARK 4.8. We might have considered a more symmetrical presentation in Definition 2.3 by allowing patterns for synthesized attributes in the right-hand sides of rules with the effect of creating forms in a configuration with patterns in their co-arguments. These patterns would express constraints on synthesized values. This extension could be acceptable if one sticks to purely centralized models. However, as soon as one wants to distribute the model on an asynchronous architecture, one cannot avoid such a constraint to be further refined due to a transformation occurring in a distant location. Then the monotony property (Proposition 4.6) is lost: A locally enabled rule can later be disabled when a constraint on a synthesized value gets a refined value. This is why we required synthesized attributes in the right-hand side of a rule to be given by plain variables in order to prohibit constraints on synthesized values.

End of Remark 4.8

It is difficult to verify input-enabledness as the whole set of accessible configurations is involved in this condition. Nevertheless one can find a sufficient condition for input enabledness, similar to the strong non-circularity of attribute grammars [9], that can be checked by a simple fixpoint computation.

DEFINITION 4.9. *Let s be a sort of a guarded attribute grammar with n inherited attributes and m synthesized attributes. We let $(j, i) \in SI(s)$ where $1 \leq i \leq n$ and $1 \leq j \leq m$ if there exists $X = s(d_1, \dots, d_n)(y_1, \dots, y_m) \in \Gamma$ where Γ is an accessible configuration and $y_j \in d_i$. If R is a rule with left-hand side $s(p_1, \dots, p_n)(u_1, \dots, u_m)$ we let $(i, j) \in IS(R)$ if there exists a variable $x \in \text{var}(R)$ such that $x \in \text{var}(d_i) \cap \text{var}(u_j)$. The guarded attribute grammar G is said to be **acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $G(s, R) = SI(s) \cup IS(R)$ is acyclic.*

PROPOSITION 4.10. *An acyclic guarded attribute grammar is input-enabled.*

PROOF. Suppose R is triggered in node X with substitution σ_{in} such that $y_j \in u_i\sigma_{in}$ then $(i, j) \in G(s, R)$. Then the fact that occur check fails for the set $\{y_j \mid 1 \leq j \leq m\}$ entails that one can find a cycle in $G(s, R)$. Q.E.D.

Relation $SI(s)$ still takes into account the whole set of accessible configurations. The following definition provides an overapproximation of this relation given by a fixpoint computation.

DEFINITION 4.11. *The **graph of local dependencies** of a rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ is the directed graph $GLD(R)$ that records the data dependencies between the occurrences of attributes given by the semantics rules. We designate the occurrences of attributes of R as follows: We let $k(i)$ (respectively $k(j)$) denote the occurrence of the i^{th} inherited attribute (resp. the j^{th} synthesized attribute) in F_k . If s is a sort with n inherited attributes and m synthesized attributes we define the relations $\overline{IS(s)}$ and $\overline{SI(s)}$ over $[1, n] \times [1, m]$ and $[1, m] \times [1, n]$ respectively as the least relations such that:*

1. *For every rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ where form F_i is of sort s_i and for every $k \in [1, \ell]$*

$$\{(j, i) \mid (k(j), k(i)) \in GLD(R)^k\} \subseteq \overline{SI(s_k)}$$

where graph $GLD(R)^k$ is given as the transitive closure of

$$GLD(R) \cup \{(0(j), 0(i)) \mid (j, i) \in \overline{SI(s_0)}\} \\ \cup \{(k'(i), k'(j)) \mid k' \in [1, \ell], k' \neq k, (i, j) \in \overline{IS(s_{k'})}\}$$

2. *For every rule $R : F_0 \rightarrow F_1 \cdots F_\ell$ where form F_i is of sort s_i*

$$\{(i, j) \mid (0(i), 0(j)) \in GLD(R)^0\} \subseteq \overline{IS(s_0)}$$

where graph $GLD(R)^0$ is given as the transitive closure of

$$GLD(R) \cup \{(k(i), k(j)) \mid k \in [1, \ell], (i, j) \in \overline{IS(s_k)}\}$$

The guarded attribute grammar G is said to be **strongly-acyclic** if for every sort s and rule R whose left-hand side is a form of sort s the graph $\overline{G(s, R)} = \overline{SI(s)} \cup IS(R)$ is acyclic.

PROPOSITION 4.12. A strongly-acyclic guarded attribute grammar is acyclic and hence input-enabled.

PROOF. The proof is analog to the proof that a strongly non-circular attribute grammar is non-circular and it goes as follows. We let $(i, j) \in IS(s)$ when $\text{var}(d_i\sigma) \cap \text{var}(y_j\sigma) \neq \emptyset$ for some form $F = s(d_1, \dots, d_n)\langle y_1, \dots, y_m \rangle$ of sort s and where σ is the substitution induced by a firing sequence starting from configuration $\{X = F\}$. Then we show by induction on the length of the firing sequence leading to the accessible configuration that $IS(s) \subseteq \overline{IS(s)}$ and $SI(s) \subseteq \overline{SI(s)}$. Q.E.D.

Note that the following two inclusions are strict

strongly-acyclic GAG \subsetneq acyclic GAG \subsetneq input enabled GAG

Indeed the reader may easily check that the guarded attribute grammar with rules

$$\begin{cases} A(x)\langle z \rangle \rightarrow B(a(x, y))\langle y, z \rangle \\ B(a(x, y))\langle x, y \rangle \rightarrow \end{cases}$$

is cyclic and input-enabled whereas guarded attribute grammar with rules

$$\begin{cases} A(x)\langle z \rangle \rightarrow B(y, x)\langle z, y \rangle \\ A(x)\langle z \rangle \rightarrow B(x, y)\langle y, z \rangle \\ B(x, y)\langle x, y \rangle \rightarrow \end{cases}$$

is acyclic but not strongly-acyclic. Attribute grammars arising from real situations are almost always strongly non-circular so that this assumption is not really restrictive. Similarly we are confident that most of the guarded attribute grammars that we shall use in practise will be input-enabled and that most of the input-enabled guarded attribute grammars are in fact strongly-acyclic. Thus most of the specifications are distributable and in most cases, this can be proved by checking strong non-circularity.

Let us conclude this section by addressing the modularity of strong-acyclicity. This property (see Def. 4.11) however was defined for autonomous guarded attribute grammars viewed as standalone applications. Here, the initial configuration is associated with a case $c = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ introduced by the external environment. And the information transmitted to the inherited attributes are given by ground terms. Even though one can imagine that these values are introduced gradually they do not depend on the values that will be returned to the subscribing variables y_1, \dots, y_m . It is indeed reasonable to assume that when a user enters a new case in the system she instantiates the inherited information and then waits for the returned values. Things go differently if the autonomous guarded attribute grammar is not a standalone application but a component in a larger specification. In that case, a call to the service provided by this component is of the form $s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ where the values transmitted to the inherited attributes may depend (directly

or indirectly) on the subscribing variables. Definition 4.11 should be amended to incorporate these dependencies and strong-acyclicity can be lost. Therefore a component which is strongly-acyclic when used as a standalone application might lose this property when it appears as an individual component of a larger specification. Thus if the component is already implemented as a collection of subcomponents distributed on an asynchronous architecture, the correctness of this distribution can be lost if the component takes part in a larger system.

To avoid this pitfall we follow a standard contract based approach, where each component can be developed independently as long as it conforms to constraints given by assume/guarantee conditions. Assuming some properties of the environment, this approach allows to preserve properties of assembled components. In our case, we show that strong-acyclicity is preserved by composition.

DEFINITION 4.13. Let s be a sort with n inherited attributes and m synthesized attributes. We let $\mathbf{IS}(s) = [1, n] \times [1, m]$ and $\mathbf{SI}(s) = [1, m] \times [1, n]$ denote the set of (potential) dependencies between inherited and synthesized attributes of s . Let G be a guarded attribute grammar with axioms s_1, \dots, s_k and terminal symbols $s'_1, \dots, s'_{k'}$. An **assume/guarantee condition** for G is a pair $(a, g) \in \mathbf{AG}(G)$ with $a \in \mathbf{SI}(s_1) \times \dots \times \mathbf{SI}(s_k) \times \mathbf{IS}(s'_1) \times \dots \times \mathbf{IS}(s'_{k'})$ and $g \in \mathbf{IS}(s_1) \times \dots \times \mathbf{IS}(s_k) \times \mathbf{SI}(s'_1) \times \dots \times \mathbf{SI}(s'_{k'})$. Equivalently it is given by the data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ for $s \in \mathbf{axioms}(G) \cup T$. The guarded attribute grammar G is strongly-acyclic w.r.t. assume/guarantee condition (a, g) if the modified fixed-point computation of Def. 4.11, where constraints $SI(s) \subseteq \overline{SI(s)}$ for $s \in \mathbf{axioms}(G)$ and $IS(s) \subseteq \overline{IS(s)}$ for $s \in T$ are added, allows to conclude strong-acyclicity with $\overline{IS(s)} \subseteq IS(s)$ for $s \in \mathbf{axioms}(G)$ and $\overline{SI(s)} \subseteq SI(s)$ for $s \in T$.

The data $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ give an over-approximation of the attribute dependencies, the so-called 'potential' dependencies, for the axioms and the terminal symbols. They define a *contract* of the guarded attribute grammar. This contract splits into *assumptions* about its environment –SI dependencies for the axioms and IS dependencies for the terminal symbols– and *guarantees* offered in return to the environment –IS dependencies for the axioms and SI dependencies for the terminal symbols. Thus strong-acyclicity of a guarded attribute grammar G w.r.t. assume/guarantee condition (a, g) means that when the environment satisfies the assume condition, grammar G is strongly-acyclic and satisfies the guarantee condition.

The following result states the modularity of strong-acyclicity.

PROPOSITION 4.14. Let $G = G_0 \oplus \dots \oplus G_p$ be a composition of guarded attribute grammars. Let $SI(s) \in \mathbf{SI}(s)$ and $IS(s) \in \mathbf{IS}(s)$ be assumptions on the (potential) dependencies between attributes where sort s ranges over the set of axioms and terminal symbols of the components G_i –thus containing also the axioms and terminal symbols of global grammar G . These constraints restrict to assume/guarantee conditions $(a_i, g_i) \in \mathbf{AG}(G_i)$ for every local grammar and for the global grammar as well: $(a, g) \in \mathbf{AG}(G)$. Then G is

strongly-acyclic w.r.t. (a, g) when each local grammar G_i is strongly-acyclic w.r.t. (a_i, g_i) .

PROOF. The fact that the fixed-point computation for the global grammar can be computed componentwise follows from the fact that for each local grammar no rule apply locally to a terminal symbol s and consequently rule 3 in Def. 4.11 never applies for s and the value $\overline{SI(s)}$ is left unmodified during the fixpoint computation, it keeps its initial value $SI(s)$. Similarly, rule 2 in Def. 4.11 never applies for an axiom s and $\overline{IS(s)}$ keeps its initial value $IS(s)$.

Q.E.D.

Conversely if the global grammar is strongly-acyclic w.r.t. some assume/guarantee condition (a, g) then the values of $\overline{SI(s)}$ and $\overline{IS(s)}$ produced at the end of the fixpoint computation allows to complement the assume/guarantee conditions with respect to which the local grammars are strongly-acyclic. The issue is how to guess some correct assume/guarantee conditions in the first place. One can imagine that some knowledge about the problem at hand can help to derive the potential attribute dependencies, and that we can use them to type the components for their future reuse in larger specifications. In many cases however there is no such dependencies and the assume/guarantee conditions are given by empty relations.

DEFINITION 4.15. A composition $G = G_0 \oplus \dots \oplus G_p$ of guarded attribute grammars is **distributable** if each local grammar (and hence also the global grammar) is strongly-acyclic w.r.t. the empty assume/guarantee condition.

This condition might seems rather restrictive but it is not. Indeed $(i, j) \notin IS(s)$ (similarly for $(i, j) \notin SI(s)$) does not mean that the j^{th} synthesized attribute does not depend on the value received by the i^{th} inherited attribute –the inherited value influences the behaviour of the component and hence has an impact on the values that will be returned in synthesized attributes. It rather says that one should not return in the value of a synthesized attribute some data directly extracted from the value of inherited attributes, which is the most common situation. The emptiness of assume/guarantee gives us a criterion for a distributable decomposition of a guarded attribute grammar: It indicates the places where a specification can safely be split into smaller pieces. We shall denote a distributable composition as:

$$G = \langle G_1, \dots, G_p \rangle$$

where $G_1 :: \%definition\ of\ G_1$

$$\vdots$$

$G_p :: \%definition\ of\ G_p$

4.2 Soundness

A specification is *sound* if every case can reach completion no matter how its execution started. Recall from Def. 4.3 that a case $c = s_0(t_1, \dots, t_n)(x_1, \dots, x_m)$ is a ground instantiation of service s_0 , an axiom of the grammar. And, an accessible configuration is any configuration accessible from

a configuration $\Gamma_0(c) = \{X_0 = c\}$ associated with a case c (an initial configuration).

DEFINITION 4.16. A configuration is **closed** if it contains only closed nodes. An autonomous guarded attribute grammar is **sound** if a closed configuration is accessible from any accessible configuration.

We consider the finite sequences $(\Gamma_i)_{0 < i \leq n}$ and the infinite sequences $(\Gamma_i)_{0 < i < \omega}$ of accessible configurations such that $\Gamma_i \rightarrow \Gamma_{i+1}$. A finite and maximal sequence is said to be **terminal**. Hence a terminal sequence leads to a configuration that enables no rule. Soundness can then be rephrased by the two following conditions.

1. Every terminal sequence leads to a closed configuration.
2. Every configuration on an infinite sequence also belongs to some terminal sequence.

Soundness can unfortunately be proved undecidable by a simple encoding of Minsky machines.

PROPOSITION 4.17. Soundness of guarded attribute grammar is undecidable.

PROOF. We consider the following presentation of the Minsky machines. We have two registers r_1 and r_2 holding integer values. Integers are encoded with the constant **zero** and the unary operator **succ**. The machine is given by a finite list of instructions $instr_i$ for $i = 1, \dots, N$ of one of the three following forms

1. **INC(r, i)**: increment register r and go to instruction i .
2. **JZDEC(r, i, j)**: if the value of register r is 0 then go to instruction i else decrement the value of the register and go to instruction j .
3. **HALT**: terminate.

We associate a Minsky machine with a guarded attribute grammar whose sorts corresponds bijectively its instructions, $S = \{s_1, \dots, s_N\}$, with the following encoding of the program instructions by rules:

1. If $instr_k = INC(r_1, i)$ then add rule
$$Inc(k, 1, i) : s_k(x, y) \rightarrow s_i(\mathbf{succ}(x), y)$$
2. If $instr_k = INC(r_2, i)$ then add rule
$$Inc(k, 2, i) : s_k(x, y) \rightarrow s_i(x, \mathbf{succ}(y))$$
3. If $instr_k = JZDEC(r_1, i, j)$ then add the rules
$$Jz(k, 1, i) : s_k(\mathbf{zero}, y) \rightarrow s_i(\mathbf{zero}, y)$$

$$Dec(k, 1, j) : s_k(\mathbf{succ}(x), y) \rightarrow s_j(x, y)$$
4. If $instr_k = JZDEC(r_2, i, j)$ then add the rules
$$Jz(k, 2, i) : s_k(x, \mathbf{zero}) \rightarrow s_i(x, \mathbf{zero})$$

$$Dec(k, 2, j) : s_k(x, \mathbf{succ}(y)) \rightarrow s_j(x, y)$$

5. If $instr_k = \text{HALT}$ then add rule

$$\text{Halt}(k) : s_k(x, y) \rightarrow$$

Since there is a unique maximal firing sequence from the initial configuration $\Gamma_0 = \{X_0 = s_1(\mathbf{zero}, \mathbf{zero})\}$ the corresponding guarded attribute grammar is sound if and only if the computation of the corresponding Minsky machine terminates. Q.E.D.

This result is not surprising as most of non-trivial properties of an expressive enough formalism are indeed undecidable. The above encoding uses very simple features of the model: All guards are of depth at most one, the system is deterministic, and there are only inherited attributes (and in fact only two of them)! This leaves no hope to find syntactic restrictions to characterize an effective subclass of sound specifications.

Even though this problem is undecidable, soundness can still be proven for a given specification using ad-hoc arguments. Furthermore we show now that a restricted form of composition of GAG, called *hierarchical composition* preserves soundness. This result allows the construction of a large class of specifications which are sound by construction.

DEFINITION 4.18 (HIERARCHICAL COMPOSITION).

A composition $G = G_0 \oplus \dots \oplus G_n$ is **hierarchical** if each GAG G_i has a unique axiom s_i , the local grammars $C_i = G_i$ for $1 \leq i \leq n$ are autonomous, and the terminal symbols of $K = G_0$ are $\{s_1, \dots, s_n\}$.

We interpret K as a *connector* that defines the possible orchestrations of the services associated with components C_1, \dots, C_n and denote $G = K(C_1, \dots, C_n)$ such a composition. In this situation, depicted in Fig. 12, the connector pro-

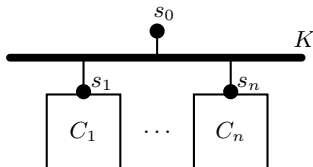


Figure 12: A hierarchical composition of GAGs

vides a global service s_0 through an orchestration of the components. For that purpose the connector can make service calls to the services s_1, \dots, s_n delivered by the components. The components are autonomous and therefore can not call their respective services. They can however communicate with each other information via attributes but only indirectly using the connector. Thus orchestration between the various components is fully encoded into the connector specification. The resulting composition, $C = K(C_1, \dots, C_n)$, is also an autonomous guarded attribute grammar and thus one can iterate this construction to obtain more complex hierarchical decompositions.

Soundness was defined in Def. 4.16 for autonomous guarded attribute grammars. We adapt this definition for a connector as follows.

DEFINITION 4.19. A guarded attribute grammar is a **connector** if it satisfies the following two conditions:

1. **Weak-Soundness.** For every accessible configuration Γ there exists a configuration accessible from Γ all of whose open nodes have sorts that are terminal symbols.
2. **Independence w.r.t. Components.** For any accessible configuration Γ , open node X , and substitution σ for variables subscribing to external services (i.e., occurring in a synthesized position in an open node of Γ whose sort is a terminal symbol), and for any rule R one has $\Gamma[R/X] \Leftrightarrow \Gamma\sigma[R/X]$, which means that a pattern of a rule never tests values produced by an external service.

Intuitively Independence w.r.t. Components means that components can exchange information through the connector but this information has no impact on the choices of rules to apply within the connector.

PROPOSITION 4.20. Let $C = K(C_1, \dots, C_n)$ be a hierarchical composition where K is a connector and the components C_1, \dots, C_n are sound. If the global grammar is input-enabled —for instance if the composition is distributable— then it is also a sound (autonomous) guarded attribute grammar.

PROOF. Let Γ be an accessible configuration of C . By Independence w.r.t. Components there exists sequences of rule applications $\Gamma_0[*]\Gamma_1[*]\Gamma$ where rules in the first sequence belong to the connector, and in the second sequence belong to the components. By Weak-Soundness, as Γ_1 is a configuration of the connector, one can find a sequence of rule applications in the connector $\Gamma_1[*]\Gamma_2$ such that the sort of an open node in Γ_2 is a terminal symbol of the connector (i.e., lies in $I(K)$). By Confluence (which follows from Input-Enabledness) there exists a configuration Γ_3 which is accessible both from Γ and from Γ_2 . The sorts of open nodes of Γ_3 are found in the components. By Soundness of the components and Monotony (which also follows from Input-Enabledness) $\Gamma_3[*]\Gamma_4$ where Γ_4 is a closed configuration. Q.E.D.

5. TOWARDS A LANGUAGE FOR THE SPECIFICATION OF GAG

In this section we introduce some syntax elements to outline a specification language for guarded attribute grammars that is expressive enough to describe realistic applications. Our purpose is not to fully design such a specification language. This would require more thorough investigations and, in particular, the implementation of some typing mechanism for the manipulated values. We only intend to introduce some syntactic sugar and constructs which allow to describe large and complex specifications in a more concise and friendlier way. This syntax is used in Section 6 where a case study is presented.

First, we introduce in Section 5.1 a functional notation for business rules, inspired from monadic programming in Haskell.

So far, a guarded attribute grammar was presented as a task rewriting system, a convenient formalism for formal manipulations. However, rewriting systems are not necessarily perceived as a handy programming notation despite their similarity with logic programming. In Section 5.2 we give the opportunity to write generic rules, namely rules that contain parameters whose instantiations can generate a potentially large set of similar rules. This is particularly useful to formalize the notion of role: When several stakeholders play a similar role they can use the same generic local grammar instantiated with their respective identities to distinguish them from one another. Section 5.3 introduces a feature that allows the designer to extend the formalism by adding combinators, a technique that can be used to customize the notations in order to derive domain specific languages adapted to the particular user needs.

5.1 A Functional Notation

In order to ease the writing of rules we introduce a syntax inspired from monadic computations in Haskell. More precisely, we restate rule

$$\begin{aligned} & \text{sort}(p_1, \dots, p_n)\langle u_1, \dots, u_m \rangle \rightarrow \\ & \quad \text{sort}_1(t_1^{(1)}, \dots, t_{n_1}^{(1)})\langle y_1^{(1)}, \dots, y_{m_1}^{(1)} \rangle \\ & \quad \dots \\ & \quad \text{sort}_k(t_1^{(k)}, \dots, t_{n_k}^{(k)})\langle y_1^{(k)}, \dots, y_{m_k}^{(k)} \rangle \end{aligned}$$

as

$$\begin{aligned} & \text{sort}(p_1, \dots, p_n) = \\ & \quad \mathbf{do} \ (y_1^{(1)}, \dots, y_{m_1}^{(1)}) \leftarrow \text{sort}_1(t_1^{(1)}, \dots, t_{n_1}^{(1)}) \\ & \quad \dots \\ & \quad (y_1^{(k)}, \dots, y_{m_k}^{(k)}) \leftarrow \text{sort}_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \\ & \quad \mathbf{return} \ (u_1, \dots, u_m) \end{aligned}$$

This functional presentation stresses out the operational purpose of business rules: Each task has an input –inherited attributes– seen as parameters and an output –synthesized attributes– seen as returned values. This notation however can confuse Haskell programmers for two reasons.

First, recall from Definition 2.3 that an input occurrence of a variable is either a variable occurring in a pattern p_i or a variable occurring as a subscription in the right-hand side of the rule or, in this alternative presentation, in the left-hand side of a *generator*

$$(y_1^{(j)}, \dots, y_{m_j}^{(j)}) \leftarrow \text{sort}_j(t_1^{(j)}, \dots, t_{n_j}^{(j)})$$

Note that, using this **do** notation, a guarded attribute grammar is left-attributed (Def. 4.5) precisely when every variable is defined before used: each output occurrence of a variable is preceded by its corresponding input occurrence. For guarded attribute grammar which are not left-attributed we thus find some variables which are used before being defined. This is incompatible with monadic programming in Haskell where the scope of a variable occurring in the left-hand side of a generator is the part of the **do** expression that follows the generator, including the return statement.

Second, a Haskell monadic expression is evaluated in *pull* mode: If the output returned by the **do** expression does not use the values of the variables defined by a given generator, then this generator is not evaluated at all. By contrast, a

rule of a guarded attribute grammar is evaluated in *push* mode: When rule is applied, we create one open node for every generator. Then users can continue to develop these nodes with the effect of gradually refining the returned values.

EXAMPLE 5.1. Consider the GAG of Example 2.2:

$$\begin{aligned} \text{Root} & : \quad \text{root}()\langle x \rangle \rightarrow \text{bin}(\text{Nil})\langle x \rangle \\ \text{Fork} & : \quad \text{bin}(x)\langle y \rangle \rightarrow \text{bin}(z)\langle y \rangle \text{bin}(x)\langle z \rangle \\ \text{Leaf}_a & : \quad \text{bin}(x)\langle \text{Cons}_a(x) \rangle \rightarrow \end{aligned}$$

Its syntactical translation into the functional notation is the following:

$$\begin{aligned} \text{Root} : \quad \text{root}() & = \mathbf{do} \ (x) \leftarrow \text{bin}(\text{Nil}) \\ & \quad \mathbf{return} \ (x) \\ \text{Fork} : \quad \text{bin}(x) & = \mathbf{do} \ (z) \leftarrow \text{bin}(x) \\ & \quad (y) \leftarrow \text{bin}(z) \\ & \quad \mathbf{return} \ (y) \\ \text{Leaf}_a \quad \text{bin}(x) & = \mathbf{do} \ \mathbf{return} \ (\text{Cons}_a(x)) \end{aligned}$$

which we write more simply as

$$\begin{aligned} \text{Root} : \quad \text{root}() & = \text{bin}(\text{Nil}) \\ \text{Fork} : \quad \text{bin}(x) & = \mathbf{do} \ (z) \leftarrow \text{bin}(x) \\ & \quad (y) \leftarrow \text{bin}(z) \\ & \quad \mathbf{return} \ (y) \\ \text{Leaf}_a \quad \text{bin}(x) & = \mathbf{return} \ (\text{Cons}_a(x)) \end{aligned}$$

using the simplification rules given below.

End of Exple 5.1

(SR₁) When the returned value is the result of the last generator, one replaces these two instructions by the last call, e.g.:

$$\mathbf{do} \ (y) \leftarrow \text{bin}(\text{Nil}) \quad \mathbf{return} \ (y) \quad \Leftrightarrow \quad \mathbf{do} \ \text{bin}(\text{Nil})$$

(SR₂) When the **do** sequence is reduced to a unique item –either a call or a **return** statement– one omits the **do** instruction, e.g.:

$$\begin{aligned} \mathbf{do} \ \text{bin}(\text{Nil}) & \quad \Leftrightarrow \quad \text{bin}(\text{Nil}) \\ \mathbf{do} \ \mathbf{return} \ (\text{Cons}_a(x)) & \quad \Leftrightarrow \quad \mathbf{return} \ (\text{Cons}_a(x)) \end{aligned}$$

5.2 Parametric Rules

It may happen that several components of a composition $G = G_1 \oplus \dots \oplus G_k$ are associated with stakeholders that play the same *role* in the system and consequently use the same set of rules. To illustrate this situation let us consider the editorial process of a scholarly journal. The Editor of the journal has to make an editorial decision about a submitted paper by resorting to scientific evaluations produced by independent reviewers. The first local grammar consists of rules governing the activities of the Editor. The other local grammars describe the activities of the various reviewers. Suppose that the role of a reviewer was previously described by a guarded attribute grammar ‘evaluate’ with an homonymous axiom corresponding to the service “making a scientific evaluation of a paper” –*It is actually convenient to name a GAG by its axiom when this axiom is unique.* All the components associated with reviewers are thus given by this specification. However the components

must have disjoint sets of non-terminal symbols so that the distribution schema described in Section 3 works properly. For that purpose, each actual reviewer is attached to a *disjoint* copy of grammar 'evaluate'. Technically, a parameter *reviewer* is added to each of the non-terminal symbols of grammar 'evaluate'. Each component is then obtained by instantiating the parameter by an identifier of the corresponding reviewer. For instance `evaluate[reviewer]` is a generic sort and an actual sort is `evaluate[Paul]` where Paul is a reviewer. This allows to distinguish activities `evaluate[Paul]` and `evaluate[Ann]` that correspond to sending a paper for evaluation to Paul and to Ann respectively. We write –using a hierarchical form of composition:

```
editorial_decision(evaluate[reviewer])  where
reviewer = Paul | Ann | ...
editorial_decision :: %Grammar editorial_decision
evaluate :: %Description of the grammar evaluate
```

This construct promotes an ordinary GAG to a generic one. At the same time, it alleviates the notations by locating the use of parameters: The sort `evaluate[reviewer]` appears in grammar 'editorial_decision' but grammar 'evaluate' simply uses sort `evaluate`. Such a construction can be iterated. For instance if the journal has several editors, one may write

```
editorial_process(editorial_decision[editor])  where
editor = Mary | Frank | ...
editorial_process :: %Grammar editorial_process
editorial_decision ::
  editorial_decision(evaluate[reviewer])  where
  reviewer = Paul | Ann | ...
  editorial_decision :: %...
  evaluate :: %...
```

The convention to name a GAG by its axiom entails some overloading of notations because in a hierarchical composition, the axiom of a global grammar coincides with the axiom of the connector. Nevertheless this overloading creates no confusion and it simplifies the notations. The above description makes it clear that the editorial process relies on editorial decisions made by editors and that such a decision depends on evaluations made by reviewers.

Parameters are instantiated by the connector. For instance a rule in grammar 'editorial_process' states that the Editor takes a decision based on evaluation reports produced by two referees. This rule can be written as:

```
Evaluate_Submission[reviewer1, reviewer2]
where not(reviewer1 = reviewer2) :
submission(article) =
  do report1 ← evaluate[reviewer1]
     report2 ← evaluate[reviewer2]
     decide(report1, report2)
```

`Evaluate_Submission[reviewer1, reviewer2]` is a *generic rule*, the corresponding actual rules are obtained by choosing values for the parameters that conform to the condition given in the **where** clause. Thus it defines as many rules as pairs of distinct reviewers. We'd rather write the above rule on

the form

```
Evaluate_Submission :
submission(article) =
  input (reviewer1, reviewer2)
  where not(reviewer1 = reviewer2)
  do report1 ← evaluate[reviewer1]
     report2 ← evaluate[reviewer2]
     decide(report1, report2)
```

The **input** clause serves to highlight those parameters of the rule that are instantiated by the user when she applies this rule at an open node associated with task `submission(article)`. The corresponding instance of the generic rule, for instance

```
Evaluate_Submission[Paul, Ann]
```

which is the actual rule selected by the user, will subsequently label the node. In this manner the information about the selection of the reviewers is stored in the artifact. Parameters in the **input** clause enable user to input any kind of data and not solely instances of roles. For instance one may find the following rule in the specification of the reviewer:

```
Accept :
evaluate(article) =
  input (msg :: String)
  do report ← review(article)
     return (Yes(msg, report))
```

With this rule the reviewer informs the Editor that he accepts to review the paper. The returned value is formed with the **Yes** constructor, witnessing acceptance, with two arguments: A complementary (optional) message and a link to the report that the reviewer commits himself to subsequently produce. In this way the specification generates an infinite set of actual rules since there exists an infinite number of potential messages –including the empty one. In practice however the parameters will correspond either to a specific role in the system –whose instantiations are finite in number– or some kind of data –a message, a report, a decision, etc.– whose values should be kept in the artifact but has no impact on the subsequent behavior of the system. Therefore it will be possible to abstract the parameter values to end up with a finite guarded attribute grammar with the same behavior.

If we expand the global grammar 'editorial_decision' the rules of grammar 'evaluate' are promoted to generic rules. In particular the above rule gives rise to the following generic rule

```
Accept :
evaluate[reviewer](article) =
  input (msg :: String)
  do report ← review[reviewer](article)
     return (Yes(msg, report))
```

Note that by contrast to parameters that appear in an **input** clause, the parameters in the left-hand side of the rule do not correspond to user choices, simply because they are already instantiated in the current configuration. For instance, in an open node associated with task `evaluate[Paul](article)`, which belongs to the active workspace of Paul, only the instance of the rule associated with Paul can apply: Mary has

no possibility to accept to review a paper that was not send to her.

All parameters of rules appear either in an **input** clause or in its left-hand side. Therefore they are always left implicate in the name of the rule.

We add the following rule to the simplification rules introduced in Sect. 5.1:

(SR₃) We omit the **do** statement if it merely returns the value(s) introduced by the preceding **input** clause:

$$\begin{array}{l} \mathbf{input}(x) \\ \mathbf{return}(x) \end{array} \Leftrightarrow \mathbf{input}(x)$$

5.3 Combinators

Since we use a variant of attribute grammars and a notation inspired from monadic computations in Haskell the question naturally arises whether one can implement our specification language as a set of monadic combinators, in the line of the Parsec library [19] for functional parsers. In the present state of things it is not at all certain that such a goal is easily achievable or even feasible. Nonetheless, some useful combinators can be introduced to tailor the specification language towards more specific application domains.

For instance, one can introduce an iteration schema given by combinator **many** defined as follows. If s is a sort with inherited and synthesized attributes of respective types a and b , **many** s stands for a new sort whose inherited and synthesized attributes are lists of elements of type a and b respectively, associated with rules

$$\begin{array}{l} \mathbf{many} s (\mathbf{Cons}(\mathit{head}, \mathit{tail})) = \mathbf{do} \quad \mathit{head}' \leftarrow s(\mathit{head}) \\ \quad \mathit{tail}' \leftarrow \mathbf{many} s (\mathit{tail}) \\ \quad \mathbf{return} (\mathbf{Cons}(\mathit{head}', \mathit{tail}')) \\ \mathbf{many} s (\mathbf{Nil}) = \mathbf{return} (\mathbf{Nil}) \end{array}$$

This is only syntactic sugar: a GAG uses a set of sorts with no *a priori* structure, but one can equip the set of sorts with a set of combinators together with a type system to constraint their usage –for instance **many** $s :: a^* \rightsquigarrow b^*$ when $s :: a \rightsquigarrow b$ – and with rules which conform to this type system.

The following example, adapted from Example 2.18, uses the above combinator to describe two recurring tasks that communicate through lazy lists:

$$\begin{array}{l} \langle s, s' \rangle :: () \rightsquigarrow () \\ \mathbf{when} \quad s :: a \rightsquigarrow b \\ \quad \quad s' :: b \rightsquigarrow a \\ \langle s, s' \rangle = \mathbf{input}(x) \mathbf{where} \quad x :: a \\ \quad \mathbf{do} \quad ys \leftarrow \mathbf{many} s (\mathbf{Cons}(x, xs)) \\ \quad \quad xs \leftarrow \mathbf{many} s' (ys) \\ \quad \mathbf{return} () \end{array}$$

The corresponding compound process uses no information from and returns no information to its surrounding environment. We can improve on this example by replacing this combinator by a (potentially infinite) set of combinators using functions as extra parameters:

$$\begin{array}{l} \langle s, s' \rangle \mathit{in} \mathit{out} :: c \rightsquigarrow d \\ \mathbf{when} \quad s :: a \rightsquigarrow b \\ \quad \quad s' :: b \rightsquigarrow a \\ \quad \quad \mathit{in} :: c \rightarrow a \\ \quad \quad \mathit{out} :: a^* \times b^* \rightarrow d \\ \langle s, s' \rangle \mathit{in} \mathit{out} = \mathbf{do} \quad ys \leftarrow \mathbf{many} s (\mathbf{Cons}(\mathit{in} x, xs)) \\ \quad \quad xs \leftarrow \mathbf{many} s' (ys) \\ \quad \quad \mathbf{return} (\mathit{out} (xs, ys)) \end{array}$$

This combinator has two parameters given by variables in and out and thus it actually generate a potentially infinite set of rules according to the actual functions used to instantiate the parameters. Still, if we assume that all such parametric combinators are totally instantiated in a global GAG specification then we guarantee that we end up with a finite specification.

For instance we may define the derived combinator:

$$\begin{array}{l} [s, s'] = \langle s, s' \rangle \mathit{id} \mathit{zip} :: a \rightsquigarrow (a \times b)^* \\ \mathbf{when} \quad s :: a \rightsquigarrow b \\ \quad \quad s' :: b \rightsquigarrow a \end{array}$$

where $\mathit{id} :: a \rightarrow a$ is the identity function and $\mathit{zip} :: a^* \times b^* \rightarrow (a \times b)^*$ is the function given by:

$$\begin{array}{l} \mathit{zip} \mathbf{Nil} ys = \mathbf{Nil} \\ \mathit{zip} xs \mathbf{Nil} = \mathbf{Nil} \\ \mathit{zip} \mathbf{Cons}(x, xs) \mathbf{Cons}(y, ys) = \mathbf{Cons}((x, y), \mathit{zip} xs ys) \end{array}$$

This combinator can equivalently be specified as

$$\begin{array}{l} [s, s'] :: a \rightsquigarrow (a \times b)^* \\ \mathbf{when} \quad s :: a \rightsquigarrow b \\ \quad \quad s' :: b \rightsquigarrow a \\ [s, s'] = \mathbf{do} \quad ys \leftarrow \mathbf{many} s (\mathbf{Cons}(x, xs)) \\ \quad \quad xs \leftarrow \mathbf{many} s' (ys) \\ \quad \quad \mathbf{return} (\mathit{zip} xs ys) \end{array}$$

However, this specification is not a GAG because the semantic rules are no longer given by plain terms but they also include some basic functions (like zip). One can impose that functions that instantiate the parameters of combinators are basic functions for list and tuples manipulations. These functions are, in any case, necessary to describe the various plumbing operations between the parametric combinators and their context of use. Since these functions are lazily evaluated it does not really affect the operational semantic of GAG.

6. A DISEASE SURVEILLANCE SYSTEM

In this section, we illustrate the model of Active workspaces based on Guarded attribute grammars with the notations introduced in Section 5 on a collaborative case management system. The real world scenario is observed from a disease surveillance system. Disease Surveillance as defined by the Centers for Disease Control and Prevention (CDC) [8] is the ongoing, systematic collection, analysis, interpretation, and dissemination of data about a health-related event for use in public health action to reduce morbidity and mortality and to improve wellbeing. It is a complex process that uses data from a plethora of sources and distributes its activities over several geographically dispersed actors with heterogeneous

profiles [6, 12, 44, 3]. Each actor plays a specific role in the system and offers a number of services needed by other stakeholders. Furthermore, the overall flow of tasks and activities is highly dependent on the available data and other contextual variables. For the purpose of the illustration, the scenario has been slightly adapted and does not therefore necessarily depict what happens in a real surveillance system.

The modeled scenario describes a situation in which three sets of actors with distinct roles (Epidemiologist, Physician, Biologist) actively participate in the surveillance and investigation of outbreaks of Influenza. An artifact in this scenario contains all the information pertaining to the treatment of a suspect case. The process starts with patient visits at a physician's office. The physician receives patients, registers the signs and symptoms, and verifies whether they correspond with those contained in the Influenza declaration criteria. If the verification is successful, he immediately declares the patient as a suspect case to the Disease Surveillance Center (DSC) (**caseDeclaration**). If the declared data contains saliva samples, the latter are sent to the biologist for laboratory analysis (**laboratoryAnalysis**). In parallel, the data is automatically analyzed by an epidemiologist (**dataAnalysis**) and eventually, outbreak alarms are produced. A number of verification tasks are run on the data and the analysis results to ascertain the alarm. The epidemiologist eventually creates a list of actions (*todo* list) that will have to be carried out by the physician to complete the alarm verification (**acmCheck**). Alongside these activities, he immediately informs Public Health Officials of the situation. Results from the laboratory analyses carried out by the biologist are used together with the results from the above checks to either confirm or revoke the outbreak alarm and produce an outbreak alert (**outbreakDecl**). Based on the outbreak characteristic data, the epidemiologist analyses the risks related to the outbreak alert and proposes appropriate counter measures.

The disease surveillance system consists of both the physicians and the Disease Surveillance Center (DSC). The latter proceeds to the analyses of the suspect cases transmitted by physicians.

```
diseaseSurveillance :: ⟨visit[physician], caseAnalysis⟩
where
  physician = Alice | Bob | ...
  visit :: % Role of a physician
  caseAnalysis ::
    caseAnalysis(laboratoryAnalysis[biologist],
                 dataAnalysis[epidemiologist])
where
  epidemiologist = Ann | Paul | ...
  biologist = Frank | Mary | ...
  caseAnalysis :: % Disease Surveillance Center
  laboratoryAnalysis :: % Role of a biologist
  dataAnalysis :: % Role of an epidemiologist
```

The case analysis is given by a hierarchical composition combining the roles of biologists and epidemiologists through an homonymous connector—the connector and the compound specification have the same axiom associated with service **caseAnalysis**.

The various components are modeled in detail in the following paragraphs. The following naming conventions are used throughout this section. Services (grammar axioms) are written in **bold**, sorts of internal (local) rules are unformatted, variable names are *italized*, and constructors are unformatted with first letter capitalized. Apart from Constructors, no other identifier has its first letter capitalized.

Role of a Physician (**visit**).

The physician receives patients, clinically examines them and if necessary, declares them as suspect cases of influenza.

```
visit(patient, alarm) =
  do (symps) ← clinicalAssessment(patient)
      () ← initialCare(symp)
      caseDeclaration(patient, symps, alarm)
```

First the physician fills out a form to report the symptoms observed in the patient. This information is a parameter of the rule (**input** clause) and thus is recorded in the artifact associated with the patient.

```
clinicalAssessment(patient) = input (symps)
```

Note that we use here the simplification rule **SR₃** meaning that the input value—the symptoms—is returned as a result of the clinical assessment. Then the physician can prescribe appropriate treatments.

```
initialCare(symp) = input (care)
                  return ()
```

Again the prescribed treatments are recorded in the artifact of the patient but this information is not returned as a result since it will not be used later.

If the symptoms correspond with the Influenza declaration criteria, the physician extracts some samples to be sent to a biologist for laboratory analysis and declares the patient as a suspect case. And he commits himself to later run some further verifications on the case—**acmCheck**—if required by the DSC, and the corresponding results—**checkRes**—are sent back to the DSC to complete the case analysis.

```
caseDeclaration(patient, symps, alarm) =
  input (samples)
  do (alarm) ← caseAnalysis (SuspectCase (patient,
                                           symps,
                                           samples),
                             checkRes)
              (checkRes) ← acmCheck(alarm)
  return ()
```

Note that this rule is not left-attributed: There are mutual dependencies between the subtasks **caseAnalysis** and **checkRes** which are executed as coroutines: the **caseAnalysis** may produce an alarm that triggers the **acmCheck**, and the **acmCheck** returns check results which are used to continue the case analysis. If the patient does not have the influenza symptoms, the following rule is used instead.

```
caseDeclaration(patient, symps, _) = return ()
```

Here we have replaced variable *alarm* by a dummy variable (the wildcard) to stress that this variable is not used in the rule—and in fact it will not be assigned a value either since

the case is not reported to the DSC and thus the analysis, from which the alarm could have been produced, is not executed for this case.

Finally if an alarm is raised –*alarm=Alarm(info, todo)*– containing analysis information together with a list of verification tasks, then the physician performs the corresponding checks on the case and transmits the results.

```
acmCheck(Alarm(info, todo)) = input (checkRes)
```

Else, the following rule applies

```
acmCheck(NoAlarm) = return (-)
```

Again the wildcard indicates a variable that is not instantiated –it is not defined by the rule– and whose value is not expected elsewhere.

Disease Surveillance Center.

If the patient is reported as a suspect case of influenza, a biologist and an epidemiologist are assigned to respectively carry out biological analyses for samples sent by the physician and to do statistical analyses and other disease surveillance related computations on the reported data in order to detect and investigate disease outbreaks. These analyses can produce an alarm.

```
caseAnalysis(SuspectCase(patient, symps, samples),
              checkRes) =
  input (bio, epi)
  do (labRes) ← laboratoryAnalysis[bio](samples)
      dataAnalysis[epi](patient, symps, labRes, checkRes)
```

Note the use of simplification rule **SR**₂ due to the fact that the result returned by **caseAnalysis** is the result of **dataAnalysis**.

Role of a Biologist (laboratoryAnalysis).

The biologist receives the samples, verifies their conditioning and runs the requested analyses. The results are returned for used in confirming or revoking the outbreak alarm. For simplicity, we suppose and only model the case where the sample is well conditioned in which case the corresponding grammar is reduced to rule

```
laboratoryAnalysis(samples) = input (labResult)
```

Role of an Epidemiologist (dataAnalysis).

This service runs a number of automated data analyses and aggregation tasks on the entire declaration database with the aim of detecting aberrations from normal behaviour (outbreak alarms). This is followed by investigation tasks to better characterize the outbreak alarms. These investigative tasks (aspecific counter measure checks) may involve superposing the alarm data with information from other sources and with contextual variables. In this case study, we limit the investigative activities to a set of queries sent to the physician who declared the current case. This service is de-

finied by the following rule,

```
dataAnalysis(patient, symps, labResult, checkResult) =
  do (ack) ← storeCaseData(patient, symps)
      automatedAnalysis(ack, labResult, checkResult)
```

The epidemiologist stores the current case data in the surveillance database.

```
storeCaseData(patient, symps) = return (Ack)
```

This triggers automated analyses on the entire database of declared suspect cases. The following rule is used if the analyses raise an alarm.

```
automatedAnalysis(Ack, labResult, checkResult) =
  input (info, todo)
  do () ← notifyAuth(info)
      () ← outbreakDecl(labResult, checkResult)
      return (Alarm(info, todo))
```

Note that the alarm is emitted in the first place, prior to notifying authorities and the outbreak declaration. And these two tasks can be performed concurrently. This clearly illustrates that the elements of a **do** body –including the **return** statement– are not necessarily executed in their order of appearance.

If on the other hand no alarm is raised the following rule is used instead.

```
automatedAnalysis(Ack, -, -) = return (NoAlarm)
```

Observe that in both versions of rule **automatedAnalysis** pattern **Ack** is checked in order to ensure that the database has been updated according to the current case. This is an illustration of side effects as discussed in observation (O1), at the end of this section.

Raised alarms are immediately notified to public health authorities.

```
notifyAuth(info) = return ()
```

As earlier stated, the alarm contains a list of queries that will need to be answered by the physician who declared the suspect case. Rich with the investigation results and the laboratory results provided respectively by the physician and the biologist, the epidemiologist runs a number of alert analyses to confirm the outbreak alarm and declare an outbreak alert. He then analyses the risks involved and the public health impact of the outbreak. This information is used to propose appropriate counter measures which he communicates to public health authorities for action.

```
outbreakDecl(labResult, checkResult) =
  input (alertInfos)
  do (risks) ← riskAnalysis(alertInfos)
      (counterM) ← defineCounterMeasures(risks)
      feedback(alertInfos, counterM)
```

```
riskAnalysis(alertInfos) = input (risks)
```

```
defineCounterMeasures(risks) = input (counterM)
```

```
feedback(alertInfos, counterM) =
  input (mail_list)
  sendFeedback(mail_list, alertInfos, counterM)
```

```
sendFeedback(mail_list, alertInfos, counterM) = return ()
```

If, conversely, the outbreak alert is not confirmed the following alternative rule is chosen.

```
outbreakDecl(labResult, checkResult) = return ()
```

Let us conclude this case study with some observations.

(O1) Some tasks may have side-effects. For instance sendFeedback forwards a message about the alert and the proposed counter measures to the email addresses given in *mail_list*. Also, storeCaseData stores the declared data in some local database on which the automated analysis is run. The automated analysis of the database produces information that guides the epidemiologist in her choice of raising an alarm or not (i.e., in choosing which rule to apply for automatedAnalysis). Such side-effects are not described in the model but they can easily be attached to rules when necessary.

(O2) Rules of the form

```
task(args) = input (results)
```

corresponding to tasks that merely return values inputted by the user can be used for incremental specifications. Indeed if the rules specifying the resolution of *task(args)* are not yet designed, one can use the above temporary rule to obtain an approximate specification that can be executed and tested even though it will require the user to manually input the expected results. Then this temporary rule can progressively be refined to obtain a new rule where the results are no longer inputted by the user but synthesized from intermediate results produced by new subtasks that are introduced for that purpose.

(O3) The given specification is sound, which can be easily verified from the fact that the underlying grammar is non-recursive: A task never calls itself directly or indirectly. This situation, which is relatively common, provides a large family of sound specifications from which many other sound specifications can be built using hierarchical composition.

7. CONCLUSION

In this paper, we have proposed a declarative model of artifact-centric collaborative systems. The key idea was to represent the workspace of a stakeholder by a set of (mind)maps associated with the services that she delivers. Each map consists of the set of artifacts created by the invocations of the corresponding service. An artifact records all the information related to the treatment of the related service call. It contains open nodes corresponding to pending tasks that require user's attention. In this manner each user has a global view of the activities in which she is involved, including all relevant information needed for the treatment of the pending tasks.

Using a variant of attribute grammars, called guarded attribute grammars, one can automate the flow of information in collaborative activities with business rules that put emphasis on user's decisions. We gave an in-depth description of this model through its syntax and its behaviour. We paid attention to two crucial properties of this model. First, the input-enabled GAG satisfy a monotony property that allows to distribute the model on an asynchronous architecture. Input-enabledness is undecidable but we have identified the sufficient condition of strongly-acyclicity that can be checked very efficiently by a fixpoint computation of an over-approximation of attribute dependencies. Second, soundness is a property that asserts that any case introduced in the system can reach completion. This property is also undecidable but we have defined a hierarchical composition of GAGs that preserves soundness and thus allows to build large specifications that are sound by construction if one starts from small components which are known to be sound.

We have introduced some notations and constructs paving the way towards an expressive and user-friendly specification language for active workspaces. Also, we have demonstrated the expressive power and exemplified the key concepts of active workspaces on a case study for a disease surveillance system.

In the rest of this section we list key features of the model and draw some future research directions.

7.1 Assessment of the Model

In a nutshell *active workspaces and guarded attribute grammars provide a modular, declarative, user-centric, data-driven, distributed and reconfigurable model of case management*. It favors flexible design and execution of business process since it possesses (to varying degrees) all four forms of *Process Flexibility* proposed in [40].

Concurrency.

The lifecycle of a business artifact is implicitly represented by the grammar productions. A production decomposes a task into new subtasks and specifies constraints between their attributes in the form of the so-called semantic rules. The subtasks may then evolve independently as long as the semantic rules are satisfied. The order of execution, which may depend on value that are computed during process execution, need not (and cannot in general) be determined statically. For that reason, this model dynamically allows maximal concurrency. In comparison, models in which the lifecycle of artifacts are represented by finite automata constrain concurrency among tasks in an artificial way.

Modularity.

The GAG approach also facilitates a modular description of business processes. For instance, when laboratory test requests are sent to the biologist, the physician needs not know about the subprocess through which the specimen will pass before results are finally produced. For instance, the service *laboratoryAnalysis* can—in a more refined specification—be modeled by a large set of rules including specimen purifi-

cation, and several biological and computational processes. However, following a top-down approach, one simply introduces an attribute in which the results should eventually be synthesized and delegate the actual production of the expected outcome to an additional set of rules. The identification of the different roles involved in the business process can also contribute to enhance modularity. Finally, some techniques borrowed from attribute grammars, like descriptive composition [20, 21], decomposition by aspects [42, 41] or higher-order attribute grammars [43, 38], may also contribute to better modular designs.

Reconfiguration.

The workflow can be reconfigured at run time: New business rules (associated with productions of the grammar) can be added to the system without disturbing the current cases. By contrast, run time reconfiguration of workflows modeled by Petri nets (or similar models) is known to be a complex issue [13, 17]. One can also add “macro rules” corresponding to specific compositions of rules. For instance if the Editor-in-chief wants to handle the evaluation of a paper, he can decide to act as an associate editor and as a referee for this particular submission. However, this means forwarding the corresponding case to himself as an associate editor and then asking himself as a referee if she is willing to write a report. A more direct way to model this decision is to encapsulate these steps in a compound macro production that bypasses the intermediate communications. More generally compound rules can be introduced for handling unusual behaviors that deviates from the nominal workflow.

Logged information.

When a case is terminated, the corresponding artifact collects all relevant information of its history. Nodes are labeled by instances of the productions that have lead to the completion of the case. Henceforth, they record the decisions (the choices among the allowed productions) together with information associated with these decisions. In the case of the editorial process, a terminated case contains the names of the referees, the evaluation reports, the editorial decision, etc. A terminated case is a tree whose branches reflect causal dependencies among subactivities used to solve a case, while abstracting from concurrent subactivities. The artifacts can be collected in a log which may be used for the purpose of process mining [39] either for process discovery (by inferring a GAG from a set of artifacts using common patterns in their tree structure) or for conformance checking (by inspection of the logs produced during simulations of a model or the executions of an actual implementation).

Distribution.

Guarded attributed grammars can easily be implemented on a distributed architecture without complex communication mechanisms –like shared memory or FIFO channels. Stakeholders in a business process own open nodes, and communicate asynchronously with other stakeholders via messages. Moreover there are no edition conflicts since each part of an artifact is edited by the unique owner of the corresponding

node. Moreover, the temporary information stored by the attributes attached to open nodes no longer exist when the case has reached completion. Closing nodes eliminates temporary information without resorting to any complex mechanism of distributed garbage collection.

7.2 Future Works

An immediate milestone is the design of a prototype of the Active Workspaces runtime environment. This prototype shall contain support tools (editor, parser, checker, simulators ...) to analyze, implement, and run GAG descriptions of AW systems. The following research directions are also considered:

Coupling GAG systems with external features.

In Section 6, we have imagined a scenario where the reported cases are stored in some local database and the analysis and investigation tasks directly access and use these data. Such implicit side-effects of rules abound –see Observation (O1) Section 6. They generally do not conflict with the GAG specification but rather complement it, in basically two ways. First, they allow to associate real-world activities with a rule, like extracting samples from a patient –caseDeclaration–, sending messages –sendFeedback–, performing verifications –acmCheck, riskAnalysis–, etc. Second, they may be used to extract information from the current artifacts to build dashboards or to feed some local database that are later used to guide the user on her choice of the rule to apply for a pending task. They may, in a more coercitive fashion, suggest a specific rule to apply or even inhibit some of the rules. Some information from dashboards or contained in a local database can also be used to populate some input parameters of a rule in place of the stakeholder. The actions of the stakeholder, namely choosing which rule to apply and the values to input in the system are left unspecified in the GAG specification –they constitute its only form on non-determinism. Side-effects can thus complement the GAG specification by providing an additional support to the stakeholder in this regard. Nonetheless, if we resort to a distant database or web services then it will be necessary to put some restrictions on the allowed queries to preserve monotony and thus to guarantee a safe distribution of the specification. Similarly we must be careful, if side-effects can inhibit some rules, that this does not jeopardize soundness. By the way, it is important to dispose of a language to describe side-effects of rules and in particular a language for making queries on active workspaces. The ideal solution would be to implement GAG as a domain specific language embedded into a general purpose language. In that case we could directly write the side-effects in the host language.

Development methodology.

We need to develop a support for the derivation of a GAG specification from a problem description. Object-oriented programming uses, for that purpose, normalized notations and diagrams for specifying the involved classes, uses cases, activities and collaborations. A modeling language for GAG should concentrate on the central concepts of the model: The artifacts, task decomposition, user’s decisions, user’s

communication. For the latter one may use concepts of speech act theory [4, 45] for classifying business rules in terms of assertions, orders, requests, commitments, etc. As far as artifacts are concerned, one can observe on the two examples of the paper (Editorial process and Disease surveillance) that we have very few completed artifacts, once the case's specific information contained in the artifacts have been abstracted, One can try to extract the business rules –task decomposition and semantic rules– starting from these archetypal artifacts and answering the following questions: What are the dependencies between data field values? Who produces these values? What information does one need to produce that value? Can one identify the conditions that justify variabilities between similar artifacts? etc. As a formalism for distributed collaborative systems the GAG model should also come with a complete method for elaborating the procedure going from a problem description, through the implementation, to the deployment on a distributed asynchronous architecture. Just as with Software Processes [37], this method will provide notations which describe how to identify relevant information (roles, data, processes ...) and propose appropriate representation tools to add expressiveness to the textual descriptions of collaborative case management systems.

Applicability and Pertinence.

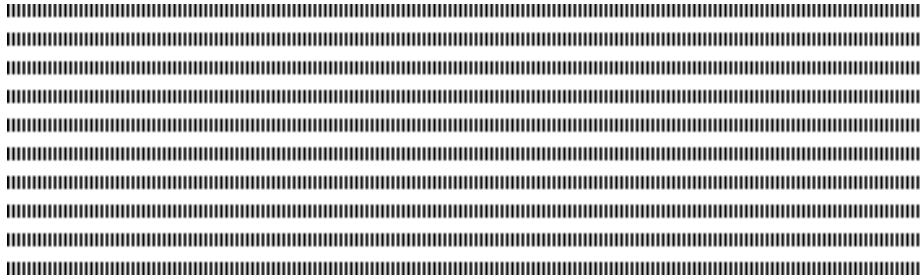
The development of case studies is very important to check the pertinence, level of applicability, and practical limitations of the GAG model. These case studies are also important to refined the specification language that we have sketched in Section 5 and to extract from practise useful concepts for a modeling language. We continue our study on Disease surveillance that we intend to effectively implement on a real situation in collaboration with epidemiologists from Centre Pasteur in Cameroon. Besides Disease surveillance, it is also useful to develop more representative case studies of distributed collaborative systems. The following two examples are representative case studies in which our model can provide advantages over existing techniques. **Reporting systems** where several stakeholders collaborate to build a report. The grammar can reflect the structure of the report, the identification of stakeholders and their respective contributions. The semantic rules implement the automatic assembly of the report from the bits provided by the distributed stakeholders. Most often, many information to be inserted in a report are already available. Semantics rules avoid redundancies and reduce workload: You *write only once* each piece of information, it is then collected in a synthesized attribute for further use. Guarded attribute grammars also avoids *email overload* –a problem that appears frequently when you have to coordinate a group of people to complete a task– since most of the communication is directly made between the active workspaces. A **distributed distance learning** which is a highly decentralized system deployed mostly in degraded environments (where Internet connection is not always available) with most stakeholders working off-line and synchronizing their activities upon establishment of an Internet connection. Also, the declarative decomposition of learning activities, which do not impose particular execution order, gives more flexibility in the de-

sign and description or learning activities and more freedom to the learner in the learning path.

8. REFERENCES

- [1] S. Abiteboul, J. Baumgarten, A. Bonifati, G. Cobena, C. Cremarenco, F. Dragan, I. Manolescu, T. Milo, and N. Preda. Managing distributed workspaces with active XML. In *VLDB*, pages 1061–1064, 2003.
- [2] S. Abiteboul, O. Benjelloun, I. Manolescu, T. Milo, and R. Weber. Active XML: A data-centric perspective on web services. In *BDA'02*, 2002.
- [3] P. Astagneau and T. Ancelle. *Surveillance épidémiologique*. Lavoisier, 2011.
- [4] J.L. Austin. *How to Do Things with Words*. Oxford Univ. Press, 1962.
- [5] K. Backhouse. A functional semantics of attribute grammars. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 2280 of *LNCS*, pages 142–157. Springer, 2002.
- [6] H. Chen, D. Zeng, and P. Yan. *Infectious Disease Informatics: Syndromic Surveillance for Public Health and Bio-Defense*. Springer, 1st edition, 2009.
- [7] L.M. Chirica and D.F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13:1–27, 1979.
- [8] World Health Organization / Centers For Disease Control. Technical Guidelines for Integrated Disease Surveillance and Response in the African Region. Technical report, WHO/CDC, Georgia, USA, 2001.
- [9] B. Courcelle and P. Franchi-Zanettacci. Attribute grammars and recursive program schemes (i) and (ii). *Theor. Comput. Sci.*, 17:163–191 and 235–257, 1982.
- [10] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. *ACM Trans. Database Syst.*, 37(3):22, 2012.
- [11] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. *Inf. Syst.*, 38(4):561–584, 2013.
- [12] V. Dato, R. Shephard, and M.M. Wagner. Outbreaks and investigations. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 13 – 26. Academic Press, Burlington, 2006.
- [13] G. De Michelis and C.A. A. Ellis. Computer supported cooperative work and Petri nets. In *Advanced Course on Petri Nets, Dagstuhl 1996*, volume 1492 of *LNCS*, pages 125–153. Springer, 1998.
- [14] P. Deransart and J. Maluszynski. Relating logic programs and attribute grammars. *J. Log. Program.*, 2(2): 119–155, 1985.
- [15] P. Deransart and J. Maluszynski. *A grammatical view of logic programming*. MIT Press, 1993.
- [16] S. Doaitse Swierstra, P.R. Azero Alcocer, and J. Saraiva. Designing and implementing combinator languages. In *Advanced Functional Programming*, pages 150–206, 1998.
- [17] C.A. Ellis and K. Keddara. MI-dews: Modeling language to support dynamic evolution within workflow

- systems. *Computer Supported Cooperative Work*, 9(3/4):293–333, 2000.
- [18] R. Eshuis, R. Hull, Y. Sun, and R. Vaculín. Splitting GSM schemas: A framework for outsourcing of declarative artifact systems. In *BPM*, vol. 8094 of *LNCS*, pages 259–274. Springer, 2013.
- [19] J. Fokker. Functional parsers. *Advanced Functional Programming, First International Spring School*, vol. 925 of *LNCS*, pages 1–23, Springer 1995.
- [20] H. Ganzinger. Increasing modularity and language-independency in automatically generated compilers. *Sci. Comput. Program.*, 3(3):223–278, 1983.
- [21] H. Ganzinger and R. Giegerich. Attribute coupled grammars. In *SIGPLAN Symposium on Compiler Construction*, pages 157–170. ACM, 1984.
- [22] Object Management Group. Business process model and notation, v 2.0. Technical report, OMG, 2011. <http://www.bpmn.org/>.
- [23] L. H elou et and A. Benveniste. Document based modeling of web services choreographies using active XML. In *ICWS*, pages 291–298. IEEE Computer Society, 2010.
- [24] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM 2008*, volume 5332 of *LNCS*, pages 1152–1163. Springer, 2008.
- [25] R. Hull, E. Damaggio, R. De Masellis, F. Fournier, M. Gupta, F.T. Heath, S. Hobson, M.H. Linehan, S. Maradugu, A. Nigam, P.N. Sukaviriya, and R. Vacul in. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. In *Fifth ACM International Conference on Distributed Event-Based Systems, DEBS 2011*, pages 51–62. ACM, 2011.
- [26] T. Johnsson. Attribute grammars as a functional programming paradigm. In *Functional Programming Languages and Computer Architecture, FPCA*, vol. 274 of *LNCS*, pages 154–173. Springer, 1987.
- [27] P. Klint, R. L ammel, and C. Verhoef. Toward an engineering discipline for grammarware. *ACM Trans. on Soft. Eng. Meth.*, 14(3): 331–380, 2005.
- [28] D.E. Knuth. Semantics of context free languages. *Mathematical System Theory*, 2(2):127–145, 1968.
- [29] N. Lohmann and K. Wolf. Artifact-centric choreographies. In *Service-Oriented Computing - 8th Int. Conf., ICSOC 2010.*, pages 32–46, 2010.
- [30] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2): 258–282, 1982.
- [31] B.H. Mayoh. Attribute grammars and mathematical semantics. *SIAM J. Comput.*, 10(3):503–518, 1981.
- [32] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Syst. J.*, 42:428–445, July 2003.
- [33] OASIS. Web services business process execution language. Tech. report, OASIS, 2007. docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.pdf.
- [34] J. Paakki. Attribute grammar paradigms - a high-level methodology in language implementation. *ACM Computing Surveys*, 27(2):196–255, 1995.
- [35] J. Saraiva and S.D. Swierstra. Generating spreadsheet-like tools from strong attribute grammars. In *Generative Programming and Component Engineering, GPCE 2003*, vol. 2830 of *LNCS*, pages 307–323. Springer, 2003.
- [36] J. Saraiva, S.D. Swierstra, and M.F. Kuiper. Functional incremental attribute evaluation. In *Compiler Construction' 2000*, vol. 1781 of *LNCS*, pages 279–294. Springer, 2000.
- [37] I. Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2011.
- [38] S.D. Swierstra and H. Vogt. Higher order attribute grammars. In *Attribute Grammars, Applications and Systems*, vol. 545 of *LNCS*, pages 256–296. Springer, 1991.
- [39] W.M.P. van der Aalst. *Process Mining - Discovery, Conformance and Enhancement of Business Processes*. Springer, 2011.
- [40] W.M.P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.
- [41] E. Van Wyk. Implementing aspect-oriented programming constructs as modular language extensions. *Sci. Comput. Program.*, 68(1):38–61, 2007.
- [42] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Compiler Construc.*, ETAPS, Grenoble, France, vol. 2304 of *LNCS*, pages 128–142, Springer 2002.
- [43] H. Vogt, S.D. Swierstra, and M.F. Kuiper. Higher-order attribute grammars. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation (PLDI)*, Portland, Oregon, pages 131–145, 1989.
- [44] M.M. Wagner, L.S. Gresham, and V. Dato. Case detection, outbreak detection, and outbreak characterization. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 27–50. Academic Press, Burlington, 2006.
- [45] T. Winograd and F. Flores. *Understanding Computer and Cognition: A new Foundation for Design*. Ablex Publishing Corporation, Norwood, New Jersey , 1986.



User Interactions in Dynamic Processes

Modeling User Interactions in Dynamic Collaborative Processes using Active Workspaces

Nsaibirni Robert Fondze Jr* — Gäetan Texier**

* LIRIMA, University of Yaounde 1
PO Box 812, Yaounde, Cameroon
Centre Pasteur of Cameroon
nsairobby@gmail.org

** Centre d'épidémiologie et de santé publique des armées (CESPA)
UMR 912 - SESSTIM - INSERM/IRD/Aix-Marseille Université
gaetan.texier@univ-amu.fr



ABSTRACT. Flexibility and change at both design- and run-time are fast becoming the Rule rather than the Exception in Business Process Models. This is attributed to the continuous advances in domain knowledge, the increase in expert knowledge, and the diverse and heterogeneous nature of contextual variables. In such processes, several users with possibly heterogeneous profiles collaborate to achieve set goals on a processes mostly designed on-the-fly. A model for such processes should thus natively support human interactions. We show in this paper how the Active Workspaces model proposed by Badouel et al. for distributed collaborative systems supports these interactions.

RÉSUMÉ. La flexibilité et la changement pendant la conception et l'exécution sont de plus en plus centrale dans les modèles des Business Process. Ceci est dû aux avancées continues des connaissances dans divers domaines, à l'augmentation des connaissances des experts, et de la nature hétérogène et multiple des variables contextuelles. Dans ces processus, plusieurs utilisateurs ayant des profils hétérogènes collaborent à des fins communs sur un processus défini progressivement. Un modèle pour de tels processus doit donc supporter nativement les interactions utilisateur. Nous montrons dans ce papier comment le modèle des Active Workspaces proposé par Badouel et al. pour la modélisation des tels processus support les interactions utilisateurs.

KEYWORDS : Collaborative Business Process, Human Interactions Patterns, Active Workspaces

MOTS-CLÉS : Processus Collaboratif, Interactions Utilisateurs, Active Workspaces



1. Introduction

Flexibility and change are fast becoming the Rule rather than the Exception in Business Process Models. As domain knowledge advances and expert knowledge increases, data and process definitions are prone to change. The need for dynamic process models is continuously being felt. Moreover, it is safe to say that dynamic process models increase user satisfaction and motivation at work, and positively influence productivity.

In [16] processes are classified as tightly-framed, loosely-framed, adhoc-framed, or unframed, depending on their predictable and repetitive nature, and on the degree of dynamism they require. The move from tightly-framed process models to unframed process models is characterized by the increasing facilities to manage uncertainty and exceptions, and the increasing influence of users and expert-knowledge in process design and enactment.

We focus on adhoc-framed and/or unframed domains, where users carry out processes in a fair degree of uncertainty[2][16] because processes cannot be completely modelled at design time either due to their large numbers or because they are highly data-centric and will have to be discovered as data is produced and as the environment evolves. In these domains, users (knowledge workers) are central to the different processes. They perform various interconnected knowledge intensive tasks and have to make complex rapid decisions on process models defined on-the-fly[2].

An example of such a domain is the disease surveillance process in public health. The process usually goes through a continuous cycle of collecting, analyzing, and dissipating information about a health condition of interest with the aim of detecting and handling unwanted events in the general population[8]. Disease surveillance is characterized as being multi-user, multi-organizational, knowledge-intensive, and time-bound[8][5]. Users and/or organizations need to collaborate and make complex rapid (timely) decisions on a semi-structured process model[2].

Like most organizational structures, a majority of national disease surveillance systems place users in a hierarchical pyramid[8]. In each level of the pyramid, users are grouped into Roles to carry out related work. Communication between the different levels of the pyramid and between the different Roles is usually through the asynchronous exchange of messages.

Our objective in this paper is to illustrate how user interactions (collaboration) in dynamic processes is supported by the Active Workspaces model[1]. We start by presenting key forms of human interactions found in business processes, then we present a purely distributed and informal specification of the Active-Workspaces model and show how it supports these interactions.

2. User Interaction Patterns

By user interaction, we mean any form of communication between a user and a computer or between two or more users via a computer[14]. Users interact in protean ways to have work done on a variety of task categories. Tasks are seen as work to be done and either originate from service calls or from work-(re)distribution in a team (work transfer and work delegation). In the following paragraphs, we describe the different ways users can interact. Our descriptions are inspired from the IBM's Business Spaces [14] that

define a human workflow attached to the underlying process model and on observations from concrete disease surveillance scenarios at the Centre Pasteur of Cameroon.

2.1. User interactions

In dynamic processes in general, users collaborate in the context of resolving specific cases. A case is a concrete instance of a business process[1]. For example, a case can comprise all tasks that will be invoked due to the arrival of a patient at a hospital or due to some outbreak alarm produced by some automated disease surveillance algorithm. One of the participating users initiates the case by instantiating the main task and providing the initially needed information. He then proceeds with the initial assignments and orchestration of tasks (work) to the other participants.

A simple description of a user's working environments could be: each of the participating users possess a work-basket which contains pending pieces of work that have been assigned to the user. In like manner, team-baskets are used to share work among a group of individuals. Task definitions contain information about the roles that have the ability to carry them out.

2.1.1. Work assignment or service request

Though users collaborate on processes in a peer-to-peer fashion, there is always a coordinating user who besides doing work is charged with initiating processes, assigning work to users, and coordinating the orchestration of the entire process. Such users exist throughout the entire process hierarchy, each managing the coordination of work that originates from him/her. Assigning work to some user (respectively to a group of users) consists in placing the work description in the user's work-basket (respectively in a group's work-basket).

2.1.2. Claiming/Releasing work

Users claim and carry out work placed in their work-baskets either based on the work-priorities or on the availability of the required input. A user can on the other hand release work placed on his/her work-basket when for some reason he is unable to carry it out.

2.1.3. Completing work

When a user claims work from his work-basket, he can either use an existing process definition to carry out the work or define a new process to do so. In both cases, he explicitly chooses the method to use and provides the required input data. For certain routine tasks, he uses a rule-based approach to define a default method to always apply.

2.1.4. Handling situations

One of the following situations may arise: a user might want to rollback and change the method he applied to resolve a task, or a user might become overbooked or unavailable or unable to complete work due to the unavailability of some input data. Such situations are handled in one or more of the following ways: undo, redo, release work, transfer work, delegate work, re-prioritize work. These strategies are applied to take into account new constraints and/or facilitate and quicken decision making.

3. User Interactions in Active-Workspaces

Explicitly described in [1], the Active-Workspaces (AW) model uses attribute grammars to represent tasks and their decomposition into sub-tasks. Inherited attributes are

used to pass data from the parent to the sub-tasks while synthesized attributes are used to return results from subtasks to parent tasks. Attributes are terms over an ordered alphabet and task triggering and execution is guarded by conditions on the inherited attributes (using First Order Logic formulas and Pattern Matching). Hence the name Guarded Attribute Grammars (GAGs) given to the underlying grammar on which Active-Workspaces are built. In this section, we will show how the Active-Workspace model supports the major aspects of user interactions presented in the previous section.

3.1. Active-Workspace: User-roles, Users, and Services

The main building block in the Active Workspaces model is the user (identified by his Active-Workspace) and collaboration between users is materialized by the exchange of services. Each user can play several roles. Services are attached to Roles and users only offer services that are attached to the Roles they play. An Active-Workspace contains:

- Guarded Attribute Grammars: A (minimal) GAG is defined for each new service in the system and copied into the workspaces of the users that offer the service (that is, users that play the role to which the service is attached). The axiom of the GAG specifies the name of the service and the productions (Business Rules) describe how this service is decomposed into subtasks. A service definition contains a unique sort s (the axiom), input variables t_i (eventually with guards), and output variables y_i .

$$s(t_1, \dots, t_n) \langle y_1, \dots, y_m \rangle$$

- Artifacts: These are process execution trees corresponding to concrete cases (work carried out by a user in his workspace). They hold data and computations pertaining to cases from their inception to their completion. The tree contains two types of nodes: Closed nodes corresponding to resolved tasks or tasks for which a resolution method has been assigned, and Open nodes corresponding tasks that await to be assigned a resolution method. Visually, an artifact is a tree with sorted nodes $X :: s$, where s is the sort of node X .

- Input Buffer: A mail box in which any service requests made to a user as well as local variables whose values are produced in distant locations are placed. In practical situations, it is divided into two; a personal inbox (work-basket) and a role-inbox (team-basket). The former contains task requests made to the user directly and the latter contains tasks made to a role the user offers which he can pick-up and execute.

- Output Buffer: Contains information produced locally and used elsewhere in the system. This includes information about distant calls to services offered by the active-workspace and distant synthesized attributes whose values will have to be produced locally in the active-workspace.

A *task* is therefore simply a guarded attribute grammar production (Business Rule). It is identified by its name (*sort*), its inherited attributes eventually with guards, its synthesized attributes, and a decomposition into subtasks showing how synthesized attributes are produced from inherited attributes. BR1 below is an example of a Business Rule.

```
BR1 :: caseAnalysis(patient, symps, antecedents, checkRes, labResult) =
do (todo, alarm, alert) ← manageAlarm(patient, symps, antecedents,
                                     labResult, checkRes)
   () ← manageAlert(alert, patient, symps, checkResult)
return(todo, alarm)
```

The above task **caseAnalysis**, extracted from the disease surveillance scenario for the monitoring of cases of Ebola[7] depicts what an Epidemiologist does when he receives a suspect case declaration (an Ebola outbreak alarm). This task receives as input information about the patient, the different checks carried out on him, and his laboratory results. It is decomposed into two subtasks `manageAlarm` and `manageAlert`, and returns two synthesized attributes *todo* and *alarm*. In like manner, we give an example of an Active-Workspace system description.

```
diseaseSurveillance :: ⟨
    consultPatient[clinician],
    laboratoryAnalysis[biologist],
    caseAnalysis[epidemiologist]
⟩

where
clinician = Alice | Bob
epidemiologist = Ann | Paul
biologist = Frank | Mary | Alice
diseaseSurveillance :: % Modelled system
consultPatient :: % Service offered by clinicians
laboratoryAnalysis :: % Service offered by biologists
caseAnalysis :: %Service offered by epidemiologists
```

Three services (`consultPatient`, `laboratoryAnalysis`, and `caseAnalysis`) are modeled in this system each offered by a distinct role (*clinician*, *biologist*, and *epidemiologist* respectively). A total of six (6) active workspaces will be generated corresponding to each of the users in the different roles. Parametric Business Rules are used in specifying Business Rules that are service calls. These simply tag the rules with the attached roles.

3.2. Requesting a service and Resolving a case

3.2.1. Requesting a service

As mentioned earlier, whatever the organizational structure, users communicate essentially by rendering and requesting services. Communication is enhanced in the Active Workspaces model using *variable subscriptions*. *Subscriptions* are equations of the form $x = u$ used to model variables x whose values u are produced at a distant site. Thus when a user calls a distant service, the synthesized attributes in the service call become subscriptions to values that will be returned by the call. Each variable has a unique defined occurrence in some workspace and may have several used occurrences elsewhere. This is enhanced using name generators that produce unique identifiers for newly created variables in each workspace.

More formally, let us consider two users: a local user identified by his active workspace AW_1 and a distant user identified by his active workspaces AW_2 . When a service call is made from AW_1 to AW_2 , the following takes place:

- $X = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ is added to the output buffer of AW_1 indicating the distant service call. This is distinguished from local calls in that there exist no defining rule for task s in AW_1 .

- $Y = s(t_1, \dots, t_n)\langle y_1, \dots, y_m \rangle$ is added to the input buffer of AW_2 , indicating that a distant service call has been made at node Y . This automatically creates a local node X and adds $Y = X$ to input buffer of AW_2 indicating where this service call is rooted in the the distant workspace.

- $x_i = u_i$ are added to the input buffer of AW_1 , indicating that variables x_i in synthesized attributes y_i subscribe to the values of distant variables u_i . In like manner, $u_i = x_i$ are added to the output buffer of AW_2 indicated variable subscriptions it will have to fulfill. These subscriptions are fulfilled incrementally, that is, values are individually returned and sent to distant subscriptions as they are produced.

3.2.2. Task orchestration bus

Resolving a Case starts from an initialisation which consists in instantiating the root node of the main service with the axiom of the GAG. This creates an artifact with a single open node. The subsequent steps (micro steps) captured in the Active Workspaces model are sanctioned either by the application of business rules to open nodes or the consumption of a fulfilled subscription from its input buffer. Either way, executing a micro step adds data to the existing system and the only ordering on these steps is imposed by their data dependencies.

A business rule R is applicable at an open node X if its left hand side matches X and if any eventual logical expression on the variables in the inherited attributes evaluates to *TRUE*. This operation of pattern matching produces a substitution σ which is a redefinition of the variables in input positions in terms of variables in output positions of both the node X and the rule R . Several rules may match the open node and the choice of which to apply is made by the user. Once a rule is chosen, node X becomes closed and new open nodes X_1, \dots, X_n are created corresponding to subtasks on the right hand side of R . At the base, these open nodes are concurrently handled with an implicit ordering imposed by variable dependences. However, it is possible to add priorities, start- and due-time to tasks and hence to nodes and recommend a certain order in the execution of these tasks. These additions can be updated at any given moment to take into account new contextual realities. Open nodes for which no applicable rule is found correspond to services that have to be requested from a distant users.

Messages received at the input buffer also update the local configuration of the Active Workspace. These messages correspond either to the reception of a service call or to the fulfillment of a subscription. The former instantiates a root node for the corresponding service in the user's workspace while the latter recursively applies the effect of the subscription up the artifact tree.

3.2.2.1. Case Transfer, Delegation, and Synchronization

Case Delegation is naturally supported through service calls and is modeled in GAGs as terminal symbols and grammar axioms. A service is offered by a role and hence by users who play the role. A user cannot call a service he offers. In other words, users cannot call services attached to roles they play. Also, each service is designed to serve a particular role. That is, only users who play that particular role can call the service. Summarily, exchange of services only occur between roles and not within roles. However, users in the same role can communicate in two ways: *Case Transfer* and *Artifact Synchronization*.

In practice, *Case Transfer* is employed as a strategy to handle situations related to user unavailability and/or inability to complete work. To transfer a case, it suffices to transfer the initial service call to the new active-workspace and update the subscriptions accordingly. This creates a new artifact on which the distant user can start working.

Case Synchronization consists in weaving artifacts of the same service enacted in different workspaces. Practically, it can be used to share information between users working on the same case (for example after a case transfer). It can be either unidirectional (a

user shares his artifact with another user) or bidirectional two users synchronize artifacts in their workspaces. This feature considers artifacts as aspects and applies an operation reminiscent to the composition of aspects in aspect oriented programming.

3.2.2.2. Evolving the Active-Workspace

If we abstract the Active-Workspace model a level or two up, it becomes evident that this model has two major separate components: a dynamic underlying guarded attribute grammar specification, and an execution engine. New business rules, services, roles, and users added to the underlying grammar are automatically taken into consideration in subsequent executions of the system. This means that users can at any moment add, remove, or change the underlying grammar and these changes are directly visible (with no retrospective effect).

These two components form a single whole to provide users with the needed flexibility in designing, executing, and managing tasks in their active workspaces which by nature are perpetually evolving.

4. Discussion and Conclusion

Dynamic processes have been at the center of BPM research recently as per these reviews: [16] and [2]. Most of these research works have focused on flexible process design with users considered as part of the external environment[3][17][4][13][10]. A few other works show how exceptions and to some extent, uncertainty are managed in dynamic processes [12][9]. These works use a set of predefined exception handlers and again do not place users at a central position. The few researchers that have carried out work on user interactions have had to define an overlying user-workflow on a predefined process workflow[14][17]. These effectively enhance user interactions by adding flexibility to process enactment but lack flexibility in process design as the process has to be defined prior to its execution.

Active workspaces provide a holistic approach to dynamic process management with users, data, and processes being the essential building blocks. This model possesses to varying degrees the different forms of process flexibility presented in [16]. This explains why it naturally supports most forms of human collaboration in dynamic processes. We have used this model to show how such interactions can be supported. It is important to note that these operations might entail coupling the Active-Workspace model with external databases, knowledge bases, time servers, process performance monitors, etc. These certainly increase an overhead on the Active-Workspace model but have no negative effect on the specifications.

5. References

- [1] Eric Badouel, Loic Helouet, Georges-edouard Kouamou, Christophe Morvan, and Robert Fondze Jr Nsaibirni. Active Workspaces : Distributed Collaborative Systems based on Guarded Attribute Grammars. *ACM SIGAPP Applied Computing Review*, 2015.
- [2] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. *Journal on Data Semantics*, pages 29–57, 2014.

- [3] R. Hull, E. Damaggio, F. Fournier, M. Gupta, Fenno Terry Heath, S. Hobson, M. H Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculín. Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA*, volume 6551 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2011.
- [4] Kunzle V, Reichert M PHILharmonicFlows: towards a framework for object-aware process management *Journal of Software Maintenance and Evolution: Research and Practice*,2011
- [5] M.M. Wagner, L.S. Gresham, and V. Dato. Chapter 3 - case detection, outbreak detection, and outbreak characterization. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 27 – 50. Academic Press, Burlington, 2006.
- [6] International Society for Disease Surveillance. Final Recommendation: Core Processes and EHR Requirements for Public Health Syndromic Surveillance. Technical report, ISDS, 2011.
- [7] R. Nsaibirni, G. Texier and GE. Kouamou. Modelling Disease Surveillance using Active Workspaces. *Conference de Recherche en Informatique (CRI), Yaounde*, 2015.
- [8] Centers For Disease Control World Health Organization. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. *Technical report, WHO/CDC, Georgia, USA* 2001.
- [9] Andrea Marrella, Massimo Mecella, Sebastian Sardina SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, {KR} 2014, Vienna, Austria, July 20-24, 2014*
- [10] Roger Atsa Etoundi, Marcel Fouda Ndjodo, and Ghislain Abessolo Aloo. A Formal Framework for Business Process Modeling. *International Journal of Computer Applications*, 13(6):27–32, 2011.
- [11] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-intensive Processes: An overview of contemporary approaches. *CEUR Workshop Proceedings*, 861:33–47, 2012.
- [12] Reichert M, Rinderle S, Kreher U, Dadam P Adaptive Process Management with ADEPT2 *ICDE*, 2005
- [13] ter Hofstede AHM, van der Aalst WMP, Adams M, Russell N Modern Business Process Automation: YAWL and its Support Environment. *Springer*, 2009
- [14] Friess Michael Business spaces for human-centric BPM , Part 1: Introduction and concepts. *IBM DeveloperWorks* 2011.
- [15] Roman Vaculín, Richard Hull, Terry Heath, Craig Cochran, Anil Nigam, and Piyawadee Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, number Edoc, pages 151–160, 2011.
- [16] Wil M. P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.
- [17] W. M. P. van der Aalst, M. Pesic, H. Schonenberg Declarative workflows: Balancing between flexibility and support *Computer Science - Research and Development*, 2009:99–113, 2009

Active-Workspaces: A Dynamic Collaborative Business Process Model for Disease Surveillance Systems

Nsaibirni Robert Fondze Jr^{1,5}, Eric Badouel², Gaëtan Texier^{3,5}, and Georges-Edouard Kouamou⁴

¹LIRIMA, University of Yaounde 1, PO Box 812, Yaounde, Cameroon

²Inria and LIRIMA, Campus de Beaulieu, 35042 Rennes, France

³Centre d'épidémiologie et de santé publique des armées (CESPA),
UMR 912 - SESSTIM - INSERM/IRD/Aix-Marseille Université

⁴LIRIMA, ENSP, PO Box 8390, Yaounde, Cameroon

⁵Centre Pasteur du Cameroun, Yaoundé, Cameroun.

Abstract—*Flexibility and change at both design- and run-time are fast becoming the Rule rather than the Exception in disease surveillance processes. This is attributed to the diversity in public health threats, to continuous advances in domain knowledge, the increase in expert knowledge, and the diverse and heterogeneous nature of contextual variables. Disease surveillance is one such processes and it is characterized by collaborative work and decision making between users with heterogeneous profiles on processes designed on-the-fly. A model for disease surveillance processes should thus natively support flexible workflow design and enactment as well as human interactions. We show in this paper how the Active Workspaces model proposed by Badouel et al. for distributed collaborative systems provides this support.*

Keywords: Disease Surveillance, Business Process Modeling, Collaborative Systems, Active-Workspaces

1. Introduction

For over twenty years, public health information systems have prospered in all medical areas and activities, in line with the advances in health informatics and related technologies. These systems are identified by the American Medical Informatics Association (AMIA) as belonging to Public Health Informatics (PHI), a specific subdomain of Health Informatics, defined as "the systematic application of information and computer science and technology to public health practice, research, and learning [23]. The scope of PHI was described as "the conceptualization, design, development, deployment, refinement, maintenance, and evaluation of communication, surveillance, information, and learning systems relevant to public health." A recent article in the AMIA yearbook of medical informatics [27] introduces a review of English-language PHI publications in Medline (2012-2014), in which authors propose main essential services such as monitoring health, supporting diagnosis, investigating outbreaks, and evaluating systems. The systems providing these services could be considered as decision support systems since it uses data, documents, knowledge and/or models to identify and solve problems and make decisions.

Concerning syndromic surveillance, defined as the continuous monitoring of public health-related information sources for early detection of adverse disease events, numerous early warning systems are currently used by experts belonging to international, national or local public health institutions. This decreases the response delays, improves effectiveness, and reduces the health impact of the outbreak. According to Chaudet et al. [26][17], outbreak identification and confirmation are managed by epidemiologists during "situation diagnosis," which consists in validating (or revoking) an alarm (signal identified as aberrant or abnormal) and transforming it into an alert (real characterized outbreak), then proposing initial countermeasures.

In health domains, known for their complexity and uncertainty, carrying out situation diagnosis implies complex decision-making processes and involves a wide range of interrelated human, biological and/or environmental activities. A disease surveillance network is thus a socio-technical system which associates geographically distant medical stakeholders (up to a few thousand people in different specialties) with dedicated systems and technical tools (telephone, satellite, digital documentation, ...) collaborating to detect and manage outbreaks [28]. More so, disease surveillance is a semi-structured [16] process which entails that only high level tasks can be clearly defined prior to process execution since most of the activities are discovered at runtime as data becomes available. This increases the complexity as users have to design and run the process-workflow on-the-fly.

Such a system in which users collaborate and share information intensively over a process model defined on-the-fly is termed a dynamic knowledge intensive system [2][11]. The modelling objective in such systems is not to completely automate the processes and their orchestrations but to provide users with expressive tools to permit them flexibly and efficiently create and run processes while making optimal use of the resources at their disposal. These tools can be grouped into four main categories:

- 1) **Tools for Real-time Iterative Workflow Construction and Orchestration:** As mentioned above, situation diagnoses for instance which is a major phase

in the syndromic surveillance process is an expert activity[17][26]. This means that the decisions and actions to be taken are determined by the expert usually based on incomplete non-pathogenic data. Thus the activity though standardised but remains highly unpredictable.

- 2) **Tools for User-Interactions:** Disease surveillance is a distributed collaborative activity (spatial and temporal) involving several stakeholders with diverse profiles[29][28]. These stakeholders interact (asynchronously) in myriad ways to find solutions to questions raised during disease surveillance[28].
- 3) **Tools for managing Exceptions and Uncertainty:** Disease surveillance data is usually described as being incomplete, non-pathogenic, and biased [17]. These are sources of uncertainty and inconclusive decision making. This uncertainty is even accentuated when attempts are made to predict future disease incidences. [30] presents uncertainty as one of the cross-cutting issues that all disease surveillance systems need to address.
- 4) **Tools to support Decision Making:** The main objective of monitoring diseases is to facilitate decision making and take timely action against public health threats[8][31][32]. In [31], PHI decision support is defined as the process of bringing relevant knowledge to bear to aid decisions involving the health and wellbeing of a population through the use of electronic information. Providing decision support is thus mandatory in all PHI information systems.

In this paper, we present an informal description of the Active-Workspaces model [1], a distributed, user-centric, and data-driven business process model built on guarded attribute grammars. Though the Active Workspaces model can be easily extended to address all of the four tools above, we limit this paper to showing how it provides support for Tools 1 and 2.

The rest of the paper is organized thus: section 2, presents related works in disease surveillance process modeling and business process modeling tools; section 3 presents an illustrative scenario; Sections 4 and 5 respectively elucidate the Active-Workspaces model with its user-centered collaborative constructs, and how the workspace can evolve. Conclusions and future works are stated in section 6.

2. Related Work

Research in public health informatics and disease surveillance in particular has focused on identifying trends/patterns in diseases, potentially viable data variables and sources, and developing novel methods of collecting, aggregating, analysing, and interpreting surveillance information. Little has been done to capture the activities, data, decision, and

collaboration schemes that are involved in disease surveillance. In [6], [5], [19] and [8] high-level steps are presented with sample activities that can be carried out at each of them. They go further to characterise the environments (pre-conditions) that favour the application of each of these activities. These pre-conditions only become satisfied at run-time thus supporting our argument for iterative process design and execution.

Futhermore, business process modelling use cases have evolved so far from models that stress on the control and coordination of tasks using state-based formalisms like automata and petri-nets [18][21][14][20][13][10], through data-centric approaches [22][25][4] that use data to dictate the orchestration of activities in a business process, to artifact centric workflows [7][3][24][15] that combine data and activities in one whole (artifacts) and use state-based [24] or declarative [3] [7][15] constructs to guide the evolution of these artifacts in a business process. These techniques however are adapted for structured-domains since they lack the required flexibility needed in disease surveillance processes and place users in the external environment.

3. Illustrative Scenario

We describe below scenarios in syndromic surveillance to better motivate the work presented in this paper.

Several users participate in this scenario: clinicians, biologists, epidemiologists, and pharmacists. We suppose that an Influenza outbreak alarm has just been raised and an epidemiologist assigned to investigate the alarm. We recall that the investigation process aims at confirming the alarm into an alert or revoking it.

The epidemiologist knows of the existence of the different actors listed above but cannot say a priori when or how he will need them during the investigation. Suppose for example that the indicator variable that produced the alarm was *pharmarcy_sales*. He will start by contacting pharmacists in the epidemic zone to ensure that the sales hike is genuine, that is, it is not caused by some commercial campaign or a similar activity. The alarm is immediately revoked if the latter is true. Otherwise, he has to investigate more. Given the high sensitivity associated with using *pharmacy_sales* as an indicator, he decides to pursue tasks that use data tightly correlated with the outbreak. In this case clinical and laboratory diagnostic data. He contacts clinicians in health districts around the epidemic zone for consultation data and runs additional analysis. He requests that patients with Influenza symptoms be contacted and samples obtained if possible and that this be carried out systematically for all new patients presenting symptoms of Influenza. He can even go ahead to request that each sample be multiplied and sent to different biologists for laboratory analysis. This especially if he possesses the required resources or if several tests need to be carried out and he wants to maximize time by spreading the tests across several laboratories.

In parallel to the activities above, he also has to manage a number of support activities such as organizing the transportation of samples from health centers to laboratories, ensuring that the laboratories possess the required reagents and equipment to run the requested tests, etc. He also has to report regularly to public health officials to help them prepare the resources to contain the potential outbreak. He continues to initiate and run activities collaboratively with other actors until he reaches a conclusion.

If on the other hand the indicator variable was different, say *school_absenteeism* or *triade_calls* or *consultation_data*, a completely different set of activities will probably be executed. Furthermore, if this task was assigned to a different epidemiologist, it is not certain that he will run the activities in the same order, or even use the same set of activities. This is because the latter and their ordering highly depend on the experience and expertise of the user and on how much he knows of his environment. Hence the Knowledge-Intensive character of surveillance systems.

This scenario shows how complex resolving a simple task might become when new data becomes available and how unpredictable the surveillance process can be. A model for such a process should therefore provide flexible constructs for building and executing process workflows on-the-fly. The fact that the process model changes is the rule and not the exception.

We also note different forms of interactions between the users and their working environments and equipment (phones, computers, etc.), and among users. For example, the epidemiologist has to interact with his work environment to accept and complete the alarm investigation request and at some point he needs to communicate with other users by sending new requests. Suppose that for some reason in the middle of the investigation, the acting epidemiologist becomes unavailable, the activities he has carried out as well as the information he has gathered will have to be transferred to the new epidemiologist. This is another form of interaction between users: synchronizing expert data.

4. Active-Workspace : User-centered Flexible and Collaborative constructs

In this section we present a succinct informal definition of the Active-Workspaces model. We lay emphasis on the properties that are required to address the two preoccupations treated in this paper. A more formal and complete description of the model is found in [1].

4.1 Active-Workspace

The Active-Workspaces (AW) model is an asynchronous cooperation model in which each participating user is assigned a workspace. A user's workspace is an arborescent (mindmap-like) structure that holds all tasks in which the user is involved as well as the data required to resolve

these tasks. The arborescent structure is reminiscent of the hierarchical organisation of tasks in which large complex tasks are broken down to small less complex ones. Each node of the mindmap has a *sort* s indicating the name of the task assigned to it. Task s can be further decomposed into subtasks s_1, \dots, s_n by applying *production* $P : s \rightarrow s_1, \dots, s_n$. A node is said to be *closed* when one such production P has been applied to it, otherwise, it is an *open node*. In the former case, the node has successors corresponding to subtasks in the right hand side of P . If the right hand side of P is empty, then node s is a *leave* of the tree. Open nodes, also called *buds*, have no successor nodes. A bud represents a *pending task* that requires the attention of the user: the bud grows when the user decides to apply a production to it. When this happens, the bud becomes a closed node associated with the production and it has n successor nodes that are newly created buds given by the subtasks s_1, \dots, s_n in the right-hand side of the production.

The hierarchical decomposition of tasks is thus not predefined but depends on decisions made by the user at each step. In disease surveillance, this is particularly useful especially during situation diagnosis. For example, faced with an Influenza outbreak alarm, an expert has to decide whether to use an approach that integrates clinical information, laboratory diagnostic information, spatial data, more profound data analysis, etc. or to just stick with an approach that combines a few of these activities. These approaches can be captured in different productions with the same sort from which the expert can choose when necessary.

Also, Active Workspaces have two main structurally independent layers: an underlying guarded attribute grammar (GAG) model and a GAG execution engine. Any changes made to the underlying grammar are directly visible to the execution engine. This means that new production rules can be added to the grammar at any time and they are immediately available for subsequent task resolution. In the example above, if the expert wants to use an approach for which no defining production exists, he can instantly create one and use it.

4.2 Collaboration and User Interactions

Each workspace is associated with at least one service rendered by the user. A service is represented by a unique sort called the *axiom* of the grammar. The particularity of this sort is that it does not appear in the right hand side of any production of the grammar. Nodes whose sorts are axioms (service nodes) are directly attached to the root node of the workspace tree. The resulting sub-tree rooted at such a node is called an *artifact*. A service call therefore instantiates a new artifact, reduced to a single bud at the root of the workspace. This artifact then develops by the application of productions until it contains no open nodes, that is, the service has been completely rendered. In a multi-user context, we model collaboration between the different

workspaces. Each workspace is associated with at least one grammar identified by its axiom and a set of productions. The sorts of a grammar are either local to the grammar (that is, they appear at the left hand side of at least one production of the grammar), or external (that is, they make reference to axioms of other grammars). Applying a production is just like in a single user scenario with the difference that a sort at the right hand side of a production which references a different grammar will be interpreted as a call to an external service. Resolving this kind of open node provokes the creation of a new artifact in the workspace of the user to whom the grammar is attached. The behaviour of the workspace remains the same as in the single user scenario but for the fact that parts of an artifact will be developed at distant sites when service calls are made.

For example, in the syndromic surveillance scenario above, the epidemiologist requests the expertise of clinicians and pharmacists to investigate the alarm. The clinician in turn requests the services of biologists to run a series of tests on extracted samples. All these interactions between the users are materialised through service calls in the Active-Workspaces model.

4.2.1 Roles

Usually several users play the same role in a system. For example in disease surveillance, there exist several clinicians, several biologists, several epidemiologists, etc. This means that these users (in the same role) are attached to the same grammars after a local renaming of the local sorts. Technically, a role is defined by a generic grammar G and we obtain the disjoint union of these grammars as follows $\oplus(r :: R)G = \biguplus_{r :: R} G[r]$ where r is a user who plays role R and $G[r]$ is the grammar obtained from G by replacing each sort (including the axiom s_0) by $s[r]$. Hence $s_0[r]$ represents service s_0 offered by r .

We note $G'\{G[r] \text{ where } r :: R\}$ the grammar made up of $\oplus(r :: R)G$ and of a grammar G' that calls this role. That is, G' will at some point need to request a service from a user in this role. In G' , we will find productions with parameters such as $P[r] : s \rightarrow s_0[r]$ expressing that when the user chooses production P to apply at an open node, he inputs a user r playing role R . The effective production is thus an instance of this generic production. We can also find in G productions of the form $P : s \rightarrow s_0[R]$ expressing that a service call is made to all users of the role R . In this case, the production has no parameters since the request will not be made to a particular user.

When a grammar needs to call several roles, we note $G\{G_1[r_1] \text{ where } r_1 :: R_1; G_2[r_2] \text{ where } r_2 :: R_2; \dots\}$ and this construction can be applied hierarchically to model chained calls as follows: $G_1\{G_2[r_2] \text{ where } r_2 :: R_2 \text{ and } G_2 = G\{G_3[r_3] \text{ where } r_3 :: R_3 \text{ and } G_3 = G\{\dots\}\}\}$. This constitution of roles is dynamic as new users can subscribe and/or un-subscribe from one or more roles at

any moment. Adding a new user to a role poses no particular difficulty since it does not modify existing workspace specifications but only modifies productions which will be called subsequently. However, removing a user from a role might become problematic if there exist in his workspace artifacts with buds. We can in such a situation either forbid the user from unsubscribing from the corresponding role, or transfer the pending artifacts to the workspace of some user of the same role. Also, as we will see later on in this paper, it is possible for a user to define new productions and extend his local grammar. This means that two users with the same role and thus with identical grammars initially might later possess different grammars. In this case, a synchronization of the two grammars is necessary before the transfer operation.

4.3 Attributes and Guards

Productions are used to structure a user's workspace. They are however not sufficient to model the interactions and data exchanges between the various tasks associated with open nodes (buds). For that purpose, we attach additional information to open nodes using *attributes*. Each sort $s \in S$ comes equipped with a set of *inherited* attributes and a set of *synthesized* attributes. Values of attributes are given by terms over a ranked alphabet. Calling a *task* is written as $(y_1, \dots, y_m) \leftarrow s(t_1, \dots, t_n)$ where the t_i 's are terms denoting the values of the inherited attributes of task and y_1, \dots, y_m are (distinct) variables subscribing to the values of its synthesized attributes. The rationale is that we invoke a task by filling in the inherited positions of the form –the inputs– and by indicating the variables that expect to receive the results returned during task execution –the subscriptions–. A (business) rule R with underlying production $s_0 \rightarrow s_1 \dots s_k$, which we note as $P[r] :: s_0 \rightarrow s_1[r] \dots s_k$, is expressed using the following notation:

$$\begin{aligned}
& s_0(p_1, \dots, p_n) = \\
& \mathbf{input}(r, z_1, \dots, z_l) \\
& \mathbf{do} \\
& \quad (y_1^{(1)}, \dots, y_{m_1}^{(1)}) \leftarrow s_1[r](t_1^{(1)}, \dots, t_{n_1}^{(1)}) \\
& \quad \dots \\
& \quad (y_1^{(k)}, \dots, y_{m_k}^{(k)}) \leftarrow s_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \\
& \mathbf{return}(u_1, \dots, u_m)
\end{aligned}$$

This functional presentation stresses out the operational purpose of business rules: Each task has an input –inherited attributes– seen as parameters and an output –synthesized attributes– seen as returned values.

- The p_i 's are patterns serving as *guards* for the rule.
- Variables z_l inside the **input** directive represent values not directly inherited from parent tasks (including users, r) and which will have to be provided by the user when the rule is chosen. This directive is omitted if no such variables exist in a rule.

- The u_j 's describe the synthesized values produced when applying the rule.
- The expressions in the right-hand side are the subtasks that will be associated with the newly created open nodes.

The variables $y_i^{(j)}$ and the variables occurring in patterns are the input variables, they are pairwise disjoint and denote respectively the information synthesized by the subtasks and the information stemming from the context of the node. The $t_i^{(j)}$ and the u_j are terms over the input variables called the *semantic rules*. They provide respectively the values of the inherited attributes of the subtasks and the values of the synthesized attributes of the main task. In this way, the values of attributes determine the rules that are applicable to resolve a task. That is, rules that are applicable at a bud.

Below is a sample grammar that models the beginning of the alarm investigation process described in Section 3. The grammar depicts a service offered by an epidemiologist. We have written in **bold** names of external sorts that make reference to other grammars in distant sites. These sorts therefore have no defining rules in this grammar. The rules are labeled **R1** to **R3** with **R1** and **R3** having parameters which will have to be filled in by the epidemiologist during execution. These parameters indicate the effective users in whose workspaces the external service requests will be made. In **R2** and **R3**, we introduced guards *FALSE* and *TRUE*. These guards automatically filter which of the two rules with sort *continue* to apply when the first task terminates. If the alarm is seen to be genuine, *TRUE*, the epidemiologist contacts a clinician sending the alarm information and a set of requests *Todos*. The result returned by this external service request is used to run additional analysis *runAnalysis* to confirm or revoke the alarm. Note that when **R3** is applied for instance, all its subtasks become buds (ready for execution). However, the *runAnalysis* bud will have to wait for the other task to complete due to variable dependences. This shows that though no predefined ordering exists between subtasks, an ordering can be introduced using variables synthesized within the subtree.

R1:

```
investigateAlarm(Alarm) =
input(pha)
do
  (real) ← contactPharm[pha](Alarm)
  (results) ← continue(real, Alarm)
return (results)
```

R2:

```
continue(FALSE, Alarm) =
return (False_Alarm)
```

R3:

```
continue(TRUE, Alarm) =
input(cli)
do
  (lab_res, Patient_data) ←
    contactClinician[cli](Alarm, Todos)
  (analysis_res) ←
    runAnalysis(lab_res, Patient_data)
return (analysis_res)
```

In some situations it is necessary that semantic rules are not given by plain terms but by more general functional expressions. This is in particular the case when one invokes a service to all individuals playing some particular role. For instance assume that the right-hand side of a rule contains a call of the form $(y) \leftarrow s[r :: R](x, y[r :: R])$. Then each individual playing role R must resolve task s to produce a synthesized result y using an inherited attribute x as well as the values y synthesized by other users of the same role. This means that to produce his results, each user uses the results produced by other users. Now, a variable y synthesized by a sort $s[r :: R]$ can only be used elsewhere in the form $y[r :: R]$, that is, a vector indexed by elements of R . Such vectors of variables cannot be used directly within terms. One might add projections to extract the variable associated with a given individual $r :: R$, which we would write $y[r]$. But in general we are not aware of a particular individual in a given role (and moreover as noted before this set of individuals can vary in time) and one is rather interested in stating conditions such as "there exist $r :: R$ such that $y[r]$ " or "for all $r :: R$, $y[r]$ ", or even "there exist at least 3 individuals $r :: R$ such that $y[r]$ ", "at least 50% of $r :: R$ verify $y[r]$ " etc. when variable y holds a boolean value. More generally one can express the semantic rules using any kind of functional expressions as long as the values of inherited attributes evaluate to terms so that they can be matched against patterns.

For example in scenario described in section 3, when a laboratory test is sought from several biologists (call them pete, bob, john, ...), and they need to each return a lab test result, *labTR*, the value returned by each of them is accessed as follows: *labTR*[pete :: biologist], *labTR*[bob :: biologist], ... It is also possible to use these in conditions like "there exist *labTR*[$r :: biologist$]" which checks if there exist any biologist who as already provided a result, "at least 3 *labTR*[$r :: biologist$]" which asserts that at least three biologist have provided results for the lab test, etc. These conditions coupled with terms are useful to drive the application of other rules at buds.

4.4 Temporal Dependencies and Constraints

Time is a critical and determining factor in user-satisfaction, cost reduction and increased productivity in business processes. In disease surveillance in particular,

timeliness is a major metric used to assert and/or evaluate the effectiveness and relevance of the process. Due to space constraints and given the extensiveness of this topic, we only present high level temporal constraints and dependencies.

We add a time-dimension to the Active-Workspaces model using the concepts defined in [34] and [33] based on Allen's Interval Algebra[35]. These works identify the following intuitive temporal constraints: *Must Start On (MSO)*, *Must Finish On (MFO)*, *As Soon As Possible (ASAP)*, *As Late As Possible (ALAP)*, *No Earlier Than (NET)* and *No Later Than (NLT)*. These constraints are attached to tasks at specification time and are used by a scheduler to control task start and end times. The specifications of the scheduler are beyond the scope of this paper. All constraints for subtasks are defined and interpreted relative to some reference point, usually the start and end times of the parent task or of sibling tasks. For instance, if data collection and data analysis are subtasks of the disease surveillance task, both subtasks can be defined to start ASAP, but the constraint on the collection task interpreted relative to the parent task and that on the analysis task interpreted relative to the collection task. The MSO and MFO constraints are strict and force the task to start or stop at exactly some time-point from the reference time. If no constraint is specified for a task, it is assumed that the task starts ASAP and finishes ASAP. Such a task is immediately executed when all necessary inputs become available and finishes as soon as all computations complete.

Also, based on the temporal constraints that exist between tasks and their data dependencies, we deduce temporal dependencies between tasks within a business rule. By temporal dependency, we mean any relationship between two tasks in which the start or end of one depends on the start or end of the other. The following four temporal dependency relationships are possible: *Start to Finish (SF)*, *Start to Start (SS)*, *Finish to Start (FS)*, and *Finish to Finish (FF)*. For instance, the data collection and data analysis tasks described in the previous paragraph have an SS relationship written $SS(\text{data collection}, \text{data analysis})$ meaning that data analysis cannot start until data collection has started. In like manner, an SF, FS, or FF relationship between two tasks S_1 and S_2 respectively means that S_2 cannot finish until S_1 starts, S_2 cannot start until S_1 finishes, and S_2 cannot finish until S_1 finishes.

Lastly, additional temporal components, Lag-Time and Lead-Time can be added to temporal dependencies to respectively account for waiting times between tasks and for overlapping tasks.

5. Workspace Evolution

The Active-Workspaces model is adapted for "Open Systems" in which the actions of users are not explicitly specified at design time. These systems are distributed and evolve dynamically with users playing a primordial role. They need to continuously design and run parts of a business

process and collaborate with each other. Even when task specifications exist, the effective actions a user undertakes (deciding which task to run, providing input data, etc.) are not specified in advance.

In section 4 we presented two ways in which a workspace can evolve. A user can either explicitly add new productions to an existing grammar or obtain productions defined by another user when their workspaces are synchronized. Also, as noted in [1], the process always interacts with external tools such as databases, email systems, time servers, etc. the so-called side-effects. These external systems complement the active-workspaces model. They allow that real world activities like extracting samples from patients, sending messages, etc. be associated to a rules. Also, these external tools can be used to extract information from enacted artifacts to build dashboards or to feed some local database that are later used to guide the user on her choice of the rule to apply for a pending task. They may, in a more coercive fashion, suggest a specific rule to apply or even inhibit some of the rules. Some information from dashboards or contained in a local database can also be used to populate some input parameters of a rule in place of the user.

6. Conclusion and Future work

In this paper, we characterized the process of monitoring diseases and health conditions of interest for unwanted events as being user-driven and data-centric. The unpredictable nature of the process further justified its knowledge-intensive characteristic. We identified four major modeling use cases that should be fulfilled by disease surveillance process modelers. The active-workspaces model can be extended to offer all four use cases. In this paper, we explicitly present how the active-workspaces model can address the first two use cases namely: dynamic workflow construction and execution, and user interactions. A prototype for the Active Workspaces model which is currently under construction will further demonstrate its pertinence and applicability.

Due to space constraints, we left out certain aspects of the Active Workspaces model which of course we will gladly add in an extended version if this paper is considered for publication. These aspects include:

- The architecture of an Active Workspace: this comprises, the underlying Guarded Attribute Grammar (GAG) engine; the Active Workspaces server that manages users and roles (adding/removing users and/or roles, subscribing/un-subscribing a user from a role), and managing communication between workspaces.
- The user interface: with visualizations of artifact trees, interfacing with external tools, enacting workflows, etc.
- GAG specifications of the key steps in the scenarios described in this paper.
- The extensive formal specification of temporal constraints and dependencies.

We are currently extending the Active Workspaces model to integrate external support for workspace construction using data mining techniques, process mining techniques, and connecting the model with a disease surveillance knowledge base. These will be necessary to demonstrate how the Active Workspaces model can be used to manage uncertainty and effective decision making, two of the use cases identified at the beginning of this paper.

Acknowledgement:

Research by the author NSAIBIRNI Robert FONDZE Jr leading to this result has been partially funded by the US Department of Health and Human Services (DHHS) through the ASIDE PROJECT (www.asideproject.org) pioneered by the Institut Pasteur of Paris and the Centre Pasteur du Cameroun.

References

- [1] Eric Badouel, Loïc Hérouët, Georges-Edouard Kouamou, Christophe Morvan, and Robert Fondze Jr Nsaibirni. Active Workspaces : Distributed Collaborative Systems based on Guarded Attribute Grammars. *ACM SIGAPP Applied Computing Review* 15(3): 6–34, 2015.
- [2] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-Intensive Processes: Characteristics, Requirements and Analysis of Contemporary Approaches. *Journal on Data Semantics*, pages 29–57, 2014.
- [3] R. Hull, E. Damaggio, F. Fournier, M. Gupta, Fenno Terry Heath, S. Hobson, M. H Linehan, S. Maradugu, A. Nigam, P. Sukaviriya, and R. Vaculín. Introducing the Guard-Stage-Milestone Approach for Specifying Business Entity Lifecycles. In *Web Services and Formal Methods - 7th International Workshop, WS-FM 2010, Hoboken, NJ, USA*, volume 6551 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 2011.
- [4] Kunzle V, Reichert M PHILharmonicFlows: towards a framework for object-aware process management *Journal of Software Maintenance and Evolution: Research and Practice*, 2011
- [5] M.M. Wagner, L.S. Gresham, and V. Dato. Chapter 3 - case detection, outbreak detection, and outbreak characterization. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 27 – 50. Academic Press, Burlington, 2006.
- [6] International Society for Disease Surveillance. Final Recommendation: Core Processes and EHR Requirements for Public Health Syndromic Surveillance. Technical report, ISDS, 2011.
- [7] R. Hull, E. Damaggio, and R. De Masellis. Business artifacts with guard-stage-milestone lifecycles: managing artifact interactions with conditions and events. ... *on Distributed event- ...*, pages 51–62, 2011.
- [8] Centers For Disease Control World Health Organization. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. *Technical report, WHO/CDC, Georgia, USA* 2001.
- [9] Andrea Marrella, Massimo Mecella, Sebastian Sardina SmartPM: An Adaptive Process Management System through Situation Calculus, IndiGolog, and Classical Planning *Principles of Knowledge Representation and Reasoning: Proceedings of the Fourteenth International Conference, {KR} 2014, Vienna, Austria, July 20-24, 2014*
- [10] Roger Atsa Etoundi, Marcel Fouda Ndjodo, and Ghislain Abessolo Aloo. A Formal Framework for Business Process Modeling. *International Journal of Computer Applications*, 13(6):27–32, 2011.
- [11] Claudio Di Ciccio, Andrea Marrella, and Alessandro Russo. Knowledge-intensive Processes: An overview of contemporary approaches. *CEUR Workshop Proceedings*, 861:33–47, 2012.
- [12] Reichert M, Rinderle S, Kreher U, Dadam P Adaptive Process Management with ADEPT2 ICDE, 2005
- [13] ter Hofstede AHM, van der Aalst WMP, Adams M, Russell N Modern Business Process Automation: YAWL and its Support Environment. *Springer*, 2009
- [14] Object Management Group Omg. Business Process Model and Notation V2.0. Technical Report December, 2010.
- [15] Roman Vaculín, Richard Hull, Terry Heath, Craig Cochran, Anil Nigam, and Piyawadee Sukaviriya. Declarative business artifact centric modeling of decision and knowledge intensive business processes. In *Proceedings - IEEE International Enterprise Distributed Object Computing Workshop, EDOC*, number Edoc, pages 151–160, 2011.
- [16] Wil M. P. van der Aalst. Business Process Management: A Comprehensive Survey. *ISRN Software Engineering*, 2013:1–37, 2013.
- [17] G Texier and Y Buisson. From epidemic outbreak detection to anticipation. *Revue d'Epidemiologie et de Sante Publique*, 2010.
- [18] W. M. P. Van Der Aalst. the Application of Petri Nets To Workflow Management. *Journal of Circuits, Systems and Computers*, 08(01):21–66, 1998.
- [19] Clementine Calba, Flavie L Goutard, Linda Hoinville, Pascal Hendrikx, Ann Lindberg, Claude Saegerman, and Marisa Peyre. Surveillance systems evaluation: a systematic review of the existing approaches. *BMC public health*, 15:448, 2015.
- [20] W. M P Van Der Aalst and a. H M Ter Hofstede. YAWL: Yet another workflow language. *Information Systems*, 30(4):245–275, 2005.
- [21] W.M.P. van der Aalst. Formalization and verification of event-driven process chains. *Information and Software Technology*, 41(10):639–650, 1999.
- [22] W.M.P. van der Aalst, R.S. Mans, and N.C. Russell. Workflow Support Using Proclerts: Divide, Interact, and Conquer. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2009.
- [23] W Yasnoff, P W O'Carroll, D Koo, R W Linkins, and E M Kilbourne. Public health informatics: improving and transforming public health in the information age. *Journal of public health management and practice* : *JPHMP*, 6(6):67–75, 2000.
- [24] Kamal Bhattacharya, Cagdas Gerede, Richard Hull, Rong Liu, and Jianwen Su. Towards Formal Analysis of Artifact-Centric Business Process Models. *Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM2007. LNCS*, 4714:288–304, 2007.
- [25] David Cohn and Richard Hull. Business Artifacts : A Data-centric Approach to Modeling Business Operations and Processes. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 32(3):1–7, 2009.
- [26] Hervé Chaudet, Pellegrin Liliane, Jean-Baptiste Meynard, Gaëtan Texier, Olivier Tournebise, Benjamin Queyriaux, and Jean-Paul Boutin. Web Services Based Syndromic Surveillance for Early Warning within French Forces. *Studies in health technology and informatics (2006)*, 124:666–671, 2006.
- [27] H P Lehmann, B E Dixon, and H Kharrazi. Public Health and Epidemiology Informatics: Recent Research and Trends in the United States. *Yearbook of medical informatics*, 10(1):199–206, 2015.
- [28] Liliane Pellegrin, Charlotte Gaudin, Nathalie Bonnardel, and Hervé Chaudet. Collaborative activities during an outbreak early warning assisted by a decision-supported system (ASTER). *Int. J. Hum. Comput. Interaction*, 26(2&3):262–277, 2010.
- [29] Daniel Zeng, Hsinchun Chen, Carlos Castillo-Chavez, and William B. Lober. *Infectious Disease Informatics and Biosurveillance*, volume 28. 2012.
- [30] Nkuchia M. M'ikanatha and John K. Iskander. Concepts and Methods in Infectious Disease Surveillance. *John Wiley & Sons, Ltd*, 2014.
- [31] Brian E Dixon, Roland E Gamache, and Shaun J Grannis. Towards public health decision support: a systematic review of bidirectional communication approaches. *Journal of the American Medical Informatics Association : JAMIA*, 20(3):577–83, 2013.
- [32] Zaruhi R Mnatsakanyan and Joseph S Lombardo. Decision Support Models for Public Health Informatics. *John Hopkins APL Technican Digest*, 27(4):332–339, 2008.
- [33] Denis Gagne and André Trudel. *Fisher, L. (Ed), 2008 BPM and Workflow Handbook*, The Temporal Perspective: Expressing Temporal Constraints and Dependencies in Process Models, pages 247-260.
- [34] Denis Gagne and André Trudel. Time-bpmm. In *Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing, CEC '09*, pages 361–367, Washington, DC, USA, 2009. IEEE Computer Society.
- [35] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, November 1983.

MODELLING DISEASE SURVEILLANCE USING ACTIVE WORKSPACES

Nsaibirni Robert Fondze Jr.^{1,2} Gaëtan Texier^{2,3} Georges-Edouard Kouamou¹

¹ University of Yaounde 1,
LIRIMA

P.O. Box 812 Yaounde, Cameroon
robert.nsaibirni@lirima.org
georges.kouamou@lirima.org

² Centre Pasteur du Cameroun
P.O. Box 1274 Yaounde, Cameroun

³UMR 912 - SESSTIM
INSERM/IRD/Aix-Marseille
Université de Marseille
gaetex1@gmail.com

ABSTRACT

Disease surveillance is characterized by the participation of several actors, geographically distributed, with diverse profiles, manipulating data from a plethora of heterogeneous sources. This, coupled with the diverse nature of public health threats and the ever increasing cognitive capabilities of the different actors account for discrepancies in the way the process is carried out in different places. Most often, the action to take at some point and how to go about it depends on the data available and on the personal expertise of the actor. In this paper, we present the Active Workspaces model, a declarative data-centric and user-driven artifact centric workflow model based on Guarded Attribute Grammars, then use it to show how workspaces of users of a disease surveillance system can be (incrementally) built and managed to enhance effective collaboration between them.

Keywords : Disease Surveillance, Business Process Management, Guarded Attribute Grammars, Active Workspaces

RESUME

La surveillance épidémiologique est caractérisé par une forte implication d'un ensemble d'acteurs avec des profils différents, se situant géographiquement à des sites différents, et manipulant des données venant de plusieurs sources hétérogènes. Ceci joint au caractère multiple et divers des menaces de santé publique et à la croissance des connaissances de ces acteurs expliquent les divergences observées dans les processus de surveillance suivies à des endroits différents. Nous présentons dans ce papier le modèle d'Active Workspaces basé sur des Grammaires Attribuées Gardées, un modèle de workflow déclaratif, centré sur les données et piloté par l'utilisateur. Nous utilisons ensuite ce modèle pour montrer comment l'espace de travail d'un acteur peut être dynamiquement construit (de façon incrémentale) et géré pour une collaboration effective entre acteurs.

Mots clés : Surveillance Epidémiologique, BPM, Grammaires Attribuées Gardées, Active Workspaces

CRI-2015, December. 14 - 15, 2015, Yaounde, Cameroon.

1. INTRODUCTION

Disease Surveillance as defined by the Centers for Disease Control and Prevention (CDC) [8] is the ongoing, systematic collection, analysis, interpretation, and dissemination of data about a health-related event for use in public health action to reduce morbidity and mortality and to improve wellbeing. Like several other data-centric, human driven processes, disease surveillance is a highly complex that, distributes its functions over a large number of individuals (with possibly heterogeneous profiles) and organizations, is characterized by an ever increasing cognitive capability of the participating individuals (actors) which entails reasoning and taking decisions based on partial or uncertain data, involves a large number of biological agents that can cause a disease as well as a myriad ways that they can present as outbreaks, and lastly, possesses numerous standards and regulations for data collection, analysis and sharing [4, 5, 7].

Several BPM modeling techniques have been used to model the disease surveillance process [4–6]. These models are highly activity-centric and assume an ideal and static behavior for the processes. They model the different phases of disease surveillance using static hierarchical trees. This rigidity in modeling coupled with the fact that specificities pertaining to the different actors and contexts are usually ignored account for the non respect of prescribed nominal procedures in running systems.

We present in this paper, an Active-Workspace construction approach that allows stakeholders to incrementally build and manage their workspaces based principally on the available data (on his/her workspace or elsewhere), on cognitive knowledge from similar tasks in the past, and on how much the stakeholder knows of his environment. This approach is based on a declarative, data centric, user driven, and distributed case management model called Guarded Attribute Grammars (GAGs) [1, 2].

The rest of the paper is organized thus: in section 2, we present the main concepts of the AW/GAG model, in section 3 we model a disease surveillance scenario using GAGs, and we conclude and state future works in section 4.

2. ACTIVE WORKSPACES

In this section, we present the key concepts that make up the Active Workspaces model. These definitions are synthesis of more formal and complete definitions given in [1].

2.1. Tasks

A **task** is a problem to be solved. It can be refined into subtasks. It is modeled by a grammar production $P : s_0 \rightarrow s_1 \cdots s_n$ expressing that task s_0 can be reduced to subtasks s_1 to s_n . For example, consulting a patient can be reduced to medically examining the patient then declaring him/her as a suspect case of some public health concern based on the observed symptoms.

$$\text{consultPatient} \rightarrow \begin{array}{l} \text{examinePatient} \\ \text{declareSuspectCase} \end{array}$$

Several Productions may exist with the same left hand side (defining the same task). The choice of which to apply corresponds to a decision made by the user.

2.2. Forms

A form is a task to which are attached additional information. Formally, a form for a task of sort s (the name of the task) is an expression $F = s(t_1, \dots, t_n)(u_1, \dots, u_m)$ where t_1, \dots, t_n (respectively u_1, \dots, u_m) are

terms over a ranked alphabet (the alphabet of attribute values) and a set of variables $\text{var}(F)$. Terms t_1, \dots, t_n give the values of the **inherited attributes** and u_1, \dots, u_m the values of the **synthesized attributes** attached to form F .

2.3. Business Rule

A business rule is a Production in which the sorts are replaced by Forms of the corresponding sort. A rule is written in either of the following forms with the second (using the functional **do** operation) used to ease their writing and stress our their operation purpose.

$$\begin{array}{l}
 s(p_1, \dots, p_n) \langle u_1, \dots, u_m \rangle \rightarrow \\
 \quad s_1(t_1^{(1)}, \dots, t_{n_1}^{(1)}) \langle y_1^{(1)}, \dots, y_{m_1}^{(1)} \rangle \\
 \quad \dots \\
 \quad s_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \langle y_1^{(k)}, \dots, y_{m_k}^{(k)} \rangle
 \end{array}
 \left|
 \begin{array}{l}
 s(p_1, \dots, p_n) = \\
 \mathbf{do} \quad (y_1^{(1)}, \dots, y_{m_1}^{(1)}) \leftarrow s_1(t_1^{(1)}, \dots, t_{n_1}^{(1)}) \\
 \quad \dots \\
 \quad (y_1^{(k)}, \dots, y_{m_k}^{(k)}) \leftarrow s_k(t_1^{(k)}, \dots, t_{n_k}^{(k)}) \\
 \mathbf{return} \quad (u_1, \dots, u_m)
 \end{array}
 \right.$$

where the p_i 's, the u_j 's, and the $t_j^{(\ell)}$'s are terms and the $y_j^{(\ell)}$'s are variables. The **defined occurrences** of variables are the occurrences of variables x_i in the patterns p_j , and the occurrences of $y_j^{(i)}$ in the left-hand parts of the **do** expression. Occurrences of variables in terms $t_i^{(j)}$, u_i , and $u^{(j)}$ are **used occurrences** of variables. Each variable has a unique defined occurrence and several used occurrences.

Parameters are added to a business rule to indicate its associated role (generic rules). For example, given that several clinicians may exist in a disease surveillance system, we create a generic rule `consultPatient[clinician]`, and the rule for each of them is obtained by instantiating *clinician* with the ID of the user. Hence, `consultPatient[Paul]`, `consultPatient[Ann]` depict the same service offered by two distinct users.

2.4. Case and Artifact

A **case** is a concrete instance of some business process. For example in disease surveillance, a case could be made up of all triggered processes due to the arrival of a patient or due to an alarm produced by some automated data analysis software. It is represented by an **artifact** which holds data and computations pertaining to the case from its inception to its completion. Visually, an artifact is a tree with sorted nodes ($X :: s$ indicates that node X is of sort s). An artifact contains closed and open nodes.

Closed node: $X = P(X_1, \dots, X_k)$ where $P : s \rightarrow s_1 \dots s_k$ is a production and $X :: s$, and $X_i :: s_i$ for $1 \leq i \leq k$. Production P is the **label** of node X and nodes X_1 to X_k are its **successor nodes**.

Open node: $X = s(t_1, \dots, t_n) \langle y_1, \dots, y_m \rangle$ where X is of sort s and t_1, \dots, t_k are terms with existing variables inherited from the predecessor node of X (inherited attributes), and y_1, \dots, y_m are newly created variables associated with its synthesized attributes,

2.5. Guarded Attribute Grammars (GAG)

Given a set of sorts S with fixed inherited and synthesized attributes, a GAG is a set of business rules $R : F_0 \rightarrow F_1, \dots, F_k$, where $F_i :: s_i$ are forms. Formally, it is a context free grammar $\mathcal{U}(G) = (N, T, A, \mathcal{P})$ where the non-terminal symbols $s \in N$ are the defined sorts, $T = S \setminus N$ is the set of terminal symbols,

$A = \mathbf{axioms}(G)$ is the set of axioms of the guarded attribute grammar, and \mathcal{P} is the set of underlying productions. Semantic rules attached to the business rules describe how synthesized attributes are computed from the inherited attributes.

2.6. Services and Active Workspace

An **active workspace** is a collection of maps, (abstract-syntax) trees used to visualize and organize tasks in which the user is involved together with the information utilized to resolved the tasks. Each map corresponds to a **service** offered by the user and each service is defined by an underlying GAG. A GAG for a service that does not call any external services is said to be autonomous. Each service is instantiated in response to specific (local or remote) requests in the context of resolving a particular case.

2.7. Configurations

A configuration Γ of an autonomous GAG is a set of artifacts in the active workspace.

2.8. Behavior of an active workspace

The behaviour of an AW expressed as the operational semantics of the underlying GAG is given by the sequence of configurations $\Gamma_0, \dots, \Gamma_n$ emanating from the application of business rules. The passage from Γ_i to Γ_j is sanctioned by the application of some rule R at an open node X , denoted as $\Gamma_i[R/X]\Gamma_j$. Rule R is said to be enabled at node X if the patterns in its LHS match with the definition of X , producing a substitution σ ($\sigma = \mathbf{match}(F_0, X)$). X becomes closed and k new open nodes (X_1, \dots, X_k) are created.

$$\begin{aligned} \Gamma_j &= \{X = R(X_1, \dots, X_k)\} \text{ where } X_1, \dots, X_k \text{ are new nodes added to } \Gamma_j \\ &\cup \{X_1 = F_1\sigma, \dots, X_k = F_k\sigma\} \text{ where } \sigma = \mathbf{match}(F_0, X) \\ &\cup \{X' = F\sigma \mid (X' = F) \in \Gamma_i \wedge X' \neq X\} \end{aligned}$$

The resolution of a case is completed when the current configuration contains no open nodes.

2.9. Composition of GAG

Let G_1, \dots, G_p be guarded attribute grammars with disjoint sets of non terminal symbols such that each terminal symbol of a grammar G_i that belongs to another grammar G_j must be an axiom of the latter. The union of all rules in each of G_i form the composed grammar.

3. MODELING DISEASE SURVEILLANCE

In this section, we model a surveillance system to monitor and investigate outbreaks of a viral hemorrhagic fever (Ebola). For simplicity, we only describe the workspaces of three categories of actors: clinicians, epidemiologists, and biologists. Patients arriving into the system are quoted to some clinician who medically examines each of them and decides whether or not to declare them as suspect cases of Ebola. Declared cases are investigated and managed by the epidemiologist and all laboratory analyses services are provided by the biologists in the system. The described process uses the alarm (suspected outbreak) / alert (confirmed outbreak) concept described in [3]. A single confirmed case of Ebola is sufficient to declare an outbreak alert.

```

diseaseSurveillance :: ⟨
    consultPatient[clinician],
    laboratoryAnalysis[biologist],
    caseAnalysis[epidemiologist]
⟩

where
clinician = Alice | Bob | ...
epidemiologist = Ann | Paul | ...
biologist = Frank | Mary | ...
diseaseSurveillance :: % Definition of the connector
consultPatient :: % Role of a clinician
laboratoryAnalysis :: % Role of a biologist
caseAnalysis :: % Role of an epidemiologist
    
```

These interactions are modeled in the following sections. We adopt the GAG naming conventions used in [1]: Services (grammar axioms) are written in **bold**, sorts of internal (local) rules are unformatted, variable names are *italized*, and constructors are unformatted with first letter capitalized. Apart from Constructors, no other identifier has its first letter capitalized.

3.1. The Clinician (Medical Doctor)

He receives patients and quoted to him, registers the symptoms and if necessary, declares them as Ebola suspect cases.

```

R1 :: consultPatient(patient) =
    do (symps, antecedents) ← examinePatient(patient)
        declareSuspectCase(SuspectCase(patient, symps, antecedents))

R2 :: examinePatient(patient) = input (symps, antecedents)
    
```

Based on the outcome of R2, the clinician decides whether or not to report the patient as a suspect case of Ebola. In the former situation, R3 is used. He quarantines the patient (R11), reports the case to the epidemiologist for further analysis (call to the external service **caseAnalysis**), prepares samples to be sent to the biologist for laboratory analysis (R4) and commits himself to carry out any supplementary checks on the patient and trace persons who had been in contact with the suspected patient.

```

R3 :: declareSuspectCase(SuspectCase(patient, symps, atdts)) =
    input (epi, samples)
    do () ← quarantinePatient(case)
        (labResult) ← requestLabAnalysis(sample)
        (checkRes) ← checkPatient(todo)
        (contactTraceRes) ← traceContacts(patient, alarm)
        (todo, alarm) ← caseAnalysis[epi](patient, symps, atdts, checkRes, labResult, contactTraceRes)
    return ()
    
```

In R4, he inputs a biologist (bio) to whom samples will be sent for analysis. We model here a situation where the biologist can either accept or refuse to carry out the laboratory analysis. The clinician therefore has the possibility to choose another biologist in the latter case. This is modeled in the in R5 and R6.

```

R4 :: requestLabAnalysis(sample) =
  input (bio)
  do (reply) ←
    laboratoryAnalysis[bio](sample)
    waitResponse(reply, sample)
    
```

```

R5 :: waitResponse(Okay(labResult), sample)
  = return (labResult)
    
```

```

R6 :: waitResponse(No(msg), sample) =
  input (bio)
  do requestLabAnalysis(sample)
    
```

R7 is used if the epidemiologist requests that additional checks be carried out.

```

R7 :: checkPatient(ToDo(todo)) = input (checkRes)
    
```

If on the other hand no additional checks are requested, the clinician invokes the following rule (R8).

```

R8 :: checkPatient(NoToDo) = return ( )
    
```

If the epidemiologist raises an alarm, then contact tracing will have to be carried out using R9. We use in this rule the **many** combinator that iteratively runs the task `traceContact` on a defined list of contacts `contactList` treating the head contact at each pass.

```

R9 :: traceContacts(Alarm(info), patient) =
  input (contactList)
  do
    many traceContact(contactList)
    
```

```

R10 :: traceContact(ContactList(contact, contacts)) =
  input (sample)
  do
    () ← quarantinePatient(contact)
    requestLabAnalysis(sample)
    
```

```

R11 :: quarantinePatient(patient) = return ( )
    
```

If no alarm is raised, then following rule is invoked instead and no contact tracing is carried out.

```

R12 :: quarantinePatient(NoAlarm, _) = return ( )
    
```

3.2. Role of a Biologist

The biologist offers the **laboratoryAnalysis** service. If for some reason he is unable to carry out the tests, he uses R13 to inform the clinician of his refusal. If however he accepts to carry out the tests, he uses R14 in which he eventually inputs the results of the tests.

```

R13 :: laboratoryAnalysis[bio](sample) =
  input (message)
  return (No(message))
    
```

```

R14 :: laboratoryAnalysis[bio](sample) =
  input (labResult)
  return (Okay(labResult))
    
```

3.3. The Epidemiologist

The epidemiologist receives and investigates all suspect cases (`caseAnalysis`). R15 is thus the root node in his/her workspace.

```

R15 :: caseAnalysis(patient, symps, antecedents, checkRes, labResult, contactTraceRes) =
  do (todo, alarm, alert) ← manageAlarm(patient, symps, antecedents, labResult, checkRes)
    () ← manageAlert(alert, patient, symps, checkResult, contactTraceRes)
    return (todo, alarm)
    
```

To investigate a case, the epidemiologist checks the plausibility of the reported case and may produce an outbreak alarm accompanied by a set of additional checks that will have to be carried out by the clinician on the patient. He then waits for the check results which he combines with the laboratory analysis results to decide whether or not there is an Ebola outbreak (declareAlert).

```
R16 :: manageAlarm(patient, symps, antecedents, labResult, checkRes) =
  do (todo, alarm) ← checkAlarmPlausibility(patient, symps, antecedents)
    (alert) ← declareAlert(patient, symps, antecedents, checkResult, labResult)
  return (todo, alarm, alert)
```

```
R17 :: checkAlarmPlausibility(patient, symps, antecedents) =
  input (todo, alarmInfos)
  return (Todo(todo), Alarm(alarmInfos))
```

```
R18 :: checkAlarmPlausibility(_, _, _) = return (NoTodo, NoAlarm)
```

To declare an outbreak alert, the epidemiologist invokes the following rule. He notifies public health authorities of the outbreak.

```
R19 :: declareAlert(patient, symps, antecedents, checkResult, labResult) =
  input (alertInfos)
  do notifyAuthorities(alertInfos, patient, symps, antecedents)
  return (Alert(alertInfos))
```

```
R20 :: notifyAuthorities(alarmInfo, patient, symps, antecedents) = return ()
```

If the investigations and laboratory analysis yield no cause to for an outbreak alert, the following rule is invoked instead.

```
R21 :: declareAlert(_, _, _, _) = return (NoAlert)
```

If an outbreak alert is produced, the epidemiologist uses the alert information, the check results, and the contact tracing results to propose appropriate counter measures.

```
R22 :: manageAlert(Alert(alertInfo), patient, symps, checkResult, contactTraceRes) =
  input (otherInfos)
  do (counterM) ← defineCounterMeasures(otherInfos, alertInfo, checkResult, contactTraceRes)
  feedback(counterM, otherInfos)
```

```
R23 :: defineCounterMeasures(otherInfos, checkResult, contactTraceRes) = input (counterM)
```

```
R24 :: feedback(counterM, otherInfos) =
  input (mailList)
  sendFeedback(mailList, otherInfos, counterM)
```

```
R25 :: sendFeedback(mailList, otherInfos, counterM) = return ()
```

If however no outbreak alert is declared, then the following rule is invoked.

```
R26 :: manageAlert(NoAlert, _, _, _) = return ()
```

3.4. Active Workspace Enactment

The possible task flows in the system are imbibed in the underlying GAG with implicit orderings given by variable dependencies. The actual choice and orchestration of tasks is done by the actor at each stage when he chooses a grammar rule to apply at an open node. GAGs are loosely tight to the enacted workspace and any modifications to the underlying GAG are visible in subsequent executions of the workspace. This means that new rules can be added and existing ones modified in a running system. These modifications however have no effect on the already enacted artifacts.

4. CONCLUSION

In this paper, we have shown that it is possible to add expressivity and flexibility to workflow management in a disease surveillance system by leaving task definition and orchestration open in a running system. The AW/GAG model we used provides highly modular, declarative, user-centric, data-driven, distributed, and reconfigurable constructs for system definition and enactment. We intend to design a domain specific language and a query language to respectively aid domain experts in running their workspaces and provide decision support dashboards with plausible feedback.

5. REFERENCES

- [1] Eric Badouel, Loic Helouet, Georges-edouard Kouamou, Christophe Morvan, and Robert Fondze Jr Nsaibirni. Active Workspaces : Distributed Collaborative Systems based on Guarded Attribute Grammars. *ACM SIGAPP Applied Computing Review*, 2015. [2](#), [5](#)
- [2] Eric Badouel, Loic Helouet, Gergers-Edouard Kouamou, and Christophe Morvan. Approach to Data-centric Case Management in a Distributed Collaborative Environment. (May):38, 2014. [2](#)
- [3] Hervé Chaudet, Pellegrin Liliane, Jean-Baptiste Meynard, Gaëtan Texier, Olivier Tournebize, Benjamin Queyriaux, and Jean-Paul Boutin. Web Services Based Syndromic Surveillance for Early Warning within French Forces. *Studies in health technology and informatics (2006)*, 124:666–671, 2006. [4](#)
- [4] H. Chen, D. Zeng, and P. Yan. *Infectious Disease Informatics: Syndromic Surveillance for Public Health and Bio-Defense*. Springer, 1st edition, 2009. [2](#)
- [5] V. Dato, R. Shephard, and M.M. Wagner. Chapter 2 - outbreaks and investigations. In M.M. Wagner, A.W. Moore, and R.M. Aryel, editors, *Handbook of Biosurveillance*, pages 13 – 26. Academic Press, Burlington, 2006. [2](#)
- [6] International Society for Disease Surveillance. Final Recommendation: Core Processes and EHR Requirements for Public Health Syndromic Surveillance. Technical report, ISDS, 2011. [2](#)
- [7] G. Texier and Y. Buisson. From epidemic outbreak detection to anticipation. *Revue d'Epidemiologie et de Sante Publique*, 2010. [2](#)
- [8] Centers For Disease Control World Health Organization. Technical Guidelines for Intergrated Disease Surveillance and Response in the African Region. Technical report, WHO/CDC, Georgia, USA, 2001. [2](#)