From monolithic architectural style to microservice one : structure-based and task-based approaches

Anfel Selmadji

THÈSE POUR OBTENIR LE GRADE DE DOCTEUR DE L'UNIVERSITE DE MONTPELLIER

En Informatique

École doctorale : Information, Structures, Systèmes

Unité de recherche LIRMM

From Monolithic Architectural Style to Microservice one : Structure-based and Task-based Approaches

Présentée par Anfel SELMADJI

Le 3 octobre 2019

Sous la direction de Abdelhak-Djamel SERIAI

Devant le jury composé de

Flavio OQUENDO, Prof, Université de Bretagne Sud	**Rapporteur (président du jury)**
Philippe COLLET, Prof, Université de Nice Sophia Antipolis	**Rapporteur**
Chirine GHEDIRA GUEGAN, Prof, Université Jean Moulin Lyon 3	**Examinatrice**
Abdelhak-Djamel SERIAI, MdC, HDR, Université de Montpellier	**Directeur**
Christophe DONY, Prof, Université de Montpellier	**Co-directeur**
Hinde Lilia BOUZIANE, MdC, Université de Montpellier	**Co-encadrante**


Microservices
Monolith
Task-based
Identification
Reengineering
Source code
Transformation
Data autonomy
Autonomous
Control flow
Software quality
Migration
Data
Java
Monolithic applications
DevOps
Reverse engineering
Microservice characteristics
PaaS
Object-oriented
Data flow
Continuous integration
Workflow
Cost
SaaS
IaaS
Quality metrics
Technology neutral
Small
Containers
Structure-based
Cloud
Pay-as-you-go
Cloud resources
Scalability
Software development
Identification
Monolith
Continuous delivery
Quality function
Focused on one function
Object-oriented

Acknowledgments

First and foremost, words cannot adequately express the gratitude I feel towards my advisor, Dr. Abdelhak-Djamel SERIAI, for all what he has done for me. The list is long. I am thankful for his continuous investment in my thesis despite the difficulties, for his patience, guidance, kindness, comprehensiveness, immense knowledge, long meeting hours even when he was exhausted,... For me, he is much more than an outstanding advisor. I am also immensely grateful to my co-advisor, Dr. Hinde Lilia BOUZIANE, for always supporting me, being extremely patient, thoughtful, and kind. My amazing co-advisor did not care how long our meetings would be, the important was that I learn how to properly tackle a problem, present things, and write them down. I am also highly thankful to my co-advisor, Prof. Christophe DONY, for his support, help as well as his constructive and valuable remarks. I am honored, blessed, and lucky to have Dr. Abdelhak-Djamel SERIAI, Dr. Hinde Lilia BOUZIANE, and Prof. Christophe DONY as advisors.

I would like to express my gratitude to my PhD committee members, Prof. Flavio OQUENDO, Prof. Philippe COLLET, and Prof. Chirine GHEDIRA GUEGAN, for having accepted to evaluate my thesis. I highly appreciate the valuable time they took from their busy schedule to do that. I am also extremely grateful for their encouragement as well as their insightful and inspiring comments, which have widened the scope of our perspectives, and will definitely improve our work.

Heartfelt appreciations go to every member of MaREL team for having welcomed me, and helped me all along. Thank you for your support, motivation, enriching discussions, and constructive remarks. I am grateful that I have known you and wish you all the best.

I would like also to express my gratitude to my family members, starting with my parents to whom I am indebted and I will never ever be able to pay that debt no matter what I do. Thank you for always believing in me, for your prayers, for your support, for your unconditional love, for your sacrifices, and for all what you have done. I am also extremely appreciative to my brother, Mouaadh, my

backbone, who have been always there for me, and gave me wise advice. Thank you for never being tired of all my questions and for turning my doubt into hope. Heartfelt appreciations go to my younger brother, Oussama, and my sister, Ritedj, for lifting up my spirit with their jokes and bright smiles. Thank you for your care, heartwarming words, and beautiful drawings.

I would like to express my appreciation to my friends for the amazing, and unforgettable time we spent together. Thank you for the laughs, for all the precious memories that I will treasure forever, and for being there for me despite the distance. I wish each and every one of you the best of luck and success in your future.

Special thanks to all the people in charge of my scholarship in the Algerian Ministry of Higher Education and Scientific Research, the University of Constantine 2, and the Algerian Consulate in Montpellier.

Finally, genuine thanks to every friend, family member, colleague, professor, and teacher who contributed to the success of this dissertation. I am lucky to have you in my life.

Résumé

Les technologies logicielles ne cessent d'évoluer pour faciliter le développement, le déploiement et la maintenance d'applications dans différents domaines. En parallèle, ces applications évoluent en continu pour garantir une bonne qualité de service et deviennent de plus en plus complexes. Cette évolution implique souvent des coûts de développement et de maintenance de plus en plus importants, auxquels peut s'ajouter une augmentation des coûts de déploiement sur des infrastructures d'exécution récentes comme le cloud. Réduire ces coûts et améliorer la qualité de ces applications sont actuellement des objectifs centraux du domaine du génie logiciel. Récemment, les microservices sont apparus comme un exemple de technologie ou style architectural favorisant l'atteinte de ces objectifs.

Un microservice peut être vu comme un service à granularité "très" fine déployable de manière indépendante et pouvant communiquer avec d'autres microservices. Les microservices permettent une reconfiguration dynamique d'une application, en ajoutant, supprimant ou remplaçant des microservices pendant son exécution. Ils semblent ainsi être bien adaptés aux pratiques DevOps qui visent à réduire les coûts depuis la création d'une application jusqu'à son déploiement et sa maintenance.

Alors que les microservices peuvent être utilisés pour développer de nouvelles applications, il existe des applications monolithiques (i.e., monolithes) construites comme une seule unité et que les propriétaires (e.g., entreprise, etc.) souhaitent maintenir et déployer sur le cloud. Dans ce cas, il est fréquent d'envisager de redévelopper ces applications à partir de rien ou d'envisager une migration vers de nouveaux styles architecturaux. Redévelopper une application ou réaliser une migration manuellement peut devenir rapidement une tâche longue, source d'erreurs et très coûteuse. Une migration automatique apparaît donc comme une solution évidente.

L'objectif principal de notre thèse est de contribuer à proposer des solutions pour l'automatisation du processus de migration d'applications monolithiques orientées objet vers des microservices. Cette migration implique deux étapes : l'identification de microservices et le packaging de ces microservices. Nous nous

focalisons sur d'identification en s'appuyant sur une analyse du code source. Nous proposons en particulier deux approches.

La première consiste à identifier des microservices en analysant les relations structurelles entre les classes du code source ainsi que les accès aux données persistantes. Dans cette approche, nous prenons aussi en compte les recommandations d'un architecte logiciel. L'originalité de ce travail peut être vue sous trois aspects. Tout d'abord, les microservices sont identifiés en se basant sur l'évaluation d'une fonction bien définie mesurant leur qualité. Cette fonction repose sur des métriques reflétant la "sémantique" du concept "microservice". Deuxièmement, les recommandations de l'architecte logiciel ne sont exploitées que lorsqu'elles sont disponibles. Enfin, deux modèles algorithmiques ont été utilisés pour partitionner les classes d'une application orientée objet en microservices : un algorithme de regroupement hiérarchique et un algorithme génétique.

La deuxième approche consiste à extraire à partir d'un code source orienté objet un workflow qui peut être utilisé en entrée de certaines approches existantes d'identification des microservices. Un workflow décrit le séquencement de tâches constituant une application suivant deux formalismes : un flot de contrôle et/ou un flot de données. L'extraction d'un workflow à partir d'un code source nécessite d'être capable de définir une correspondance entre les concepts du mon-de objet et ceux d'un workflow.

Pour valider nos deux approches, nous avons implémenté deux prototypes et mené des expérimentations sur plusieurs cas d'étude. Les microservices identifiés ont été évalués qualitativement et quantitativement. Les workflows obtenus ont été évalués manuellement sur un jeu de tests. Les résultats obtenus montrent respectivement la pertinence des microservices identifiés et l'exactitude des workflows obtenus.

Mots-clés : Monolithe, orienté objet, microservice, migration logicielle, réingénierie logicielle, qualité logicielle, métrique de qualité, workflow, flot de données, flot de contrôle, cloud.

Abstract

Software technologies are constantly evolving to facilitate the development, deployment, and maintenance of applications in different areas. In parallel, these applications evolve continuously to guarantee an adequate quality of service, and they become more and more complex. Such evolution often involves increased development and maintenance costs, that can become even higher when these applications are deployed in recent execution infrastructures such as the cloud. Nowadays, reducing these costs and improving the quality of applications are main objectives of software engineering. Recently, microservices have emerged as an example of a technology or architectural style that helps to achieve these objectives.

A microservice can be seen as a small, independently deployable service that can communicate with other microservices. Microservices allow dynamic reconfiguration of an application by adding, removing, or replacing microservices during its execution. They also seem to be well-adapted to DevOps practices that aim to reduce costs from the creation of an application to its deployment, and maintenance.

While microservices can be used to develop new applications, there are monolithic ones (i.e., monoliths) built as a single unit and their owners (e.g., companies, etc.) want to maintain and deploy them in the cloud. In this case, it is common to consider rewriting these applications from scratch or migrating them towards recent architectural styles. Rewriting an application or migrating it manually can quickly become a long, error-prone, and expensive task. An automatic migration appears as an evident solution.

The ultimate aim of our dissertation is contributing to automate the migration of monolithic Object-Oriented (OO) applications to microservices. This migration consists of two steps: microservice identification and microservice packaging. We focus on microservice identification based on source code analysis. Specifically, we propose two approaches.

The first one identifies microservices from the source code of a monolithic

OO application relying on code structure, data accesses, and software architect recommendations. The originality of our approach can be viewed from three aspects. Firstly, microservices are identified based on the evaluation of a well-defined function measuring their quality. This function relies on metrics reflecting the "semantics" of the concept "microservice". Secondly, software architect recommendations are exploited only when they are available. Finally, two algorithmic models have been used to partition the classes of an OO application into microservices: clustering and genetic algorithms.

The second approach extracts from an OO source code a workflow that can be used as an input of some existing microservice identification approaches. A workflow describes the sequencing of tasks constituting an application according to two formalisms: control flow and /or data flow. Extracting a workflow from source code requires the ability to map OO concepts into workflow ones.

To validate both approaches, we implemented two prototypes and conducted experiments on several case studies. The identified microservices have been evaluated qualitatively and quantitatively. The extracted workflows have been manually evaluated relying on test suites. The obtained results show respectively the relevance of the identified microservices and the correctness of the extracted workflows.

Keywords: Monolith, object-oriented, microservice, software migration, software reengineering, software quality, quality metrics, workflow, data flow, control flow, cloud.

Résumé étendu

Au fil des ans, les technologies évoluent pour permettre un développement et un déploiement de logiciels plus efficaces et plus faciles, ainsi que toute activité ultérieure à leur livraison, telle que la maintenance. Des exemples de technologies récentes qui permettent cela et attirent l'attention des chercheurs ainsi que des industriels, sont le cloud computing, le DevOps et les microservices.

Pour suivre le rythme des avancées technologiques, les applications existantes développées en utilisant des technologies obsolètes peuvent être réécrites à partir de zéro ou migrées vers des technologies plus récentes. Notre travail contribue à la migration des applications monolithiques existantes vers des microservices afin de les adapter à la fois au cloud computing et au DevOps.

Contexte

Cloud et DevOps

Le cloud computing est un modèle permettant l'accès réseau à un pool partagé de ressources informatiques configurables telles que l'espace de stockage, les serveurs, les applications, les services, etc. [101]. Les clients peuvent à la demande allouer et libérer rapidement ces ressources avec un minimum d'effort de gestion et de communication des fournisseurs de services [101]. Le cloud se caractérise par sa haute disponibilité, son élasticité, sa scalabilité et sa flexibilité des coûts. Il fournit une variété de services qui peuvent être classés dans une hiérarchie de termes en tant que service [127]. Une catégorisation simplifiée divise les services du cloud en trois catégories: SaaS (logiciel en tant que service), PaaS (plateforme en tant que service) et IaaS (infrastructure en tant que service). Le SaaS est un paradigme de livraison de logiciels, dans lequel les fournisseurs de services développent des logiciels et les distribuent via Internet [55]. PaaS offre des plateformes pour le développement et le déploiement d'applications dans le

cloud en utilisant des langages de programmation, des bibliothèques, des outils, etc. [57]. Les fournisseurs IaaS offrent des ressources de traitement, de stockage, de réseau et autres ressources informatiques fondamentales permettant aux clients de déployer et d'exécuter leurs logiciels [51, 101]. Cela peut être réalisé en allouant des machines virtuelles (VM) personnalisées avec une capacité de CPU, de mémoire, de stockage et de bande passante prédéfinie [127]. Les clients peuvent allouer / libérer des machines virtuelles de manière élastique, et ils sont facturés en fonction de leur utilisation. SalesForce [5], Google App Engine [2], Amazon EC2 [3] et Microsoft Azure [4] sont parmi les principaux fournisseurs de services dans le cloud.

Grâce à ses caractéristiques, de nos jours, le cloud attire l'attention des chercheurs et des industriels. Certains chercheurs le considèrent comme une plateforme de développement et d'exécution des applications scientifiques massivement scalables [105]. De plus, au lieu d'investir d'avance dans une infrastructure sur-provisionnée, ils ne paient que pour ce qu'ils utilisent. Même les entreprises sont attirées par le cloud en raison de sa scalabilité et de sa flexibilité des coûts [74]. Ils le voient comme une stratégie commerciale opportuniste leur permettant de rivaliser avec d'autres entreprises et d'atteindre leurs objectifs commerciaux [34, 17, 16], en particulier lorsqu'il est fusionné avec DevOps efficace.

DevOps (**Dév**eloppement et **Op**ération**s**) [26] est un ensemble de pratiques visant à améliorer la collaboration entre les équipes de développement (par exemple, programmation, tests, etc.) et les équipes d'opérations (par exemple, déploiement, surveillance, etc.) pour réduire le temps de mise sur le marché. De plus, ces pratiques garantissent le maintien de la qualité du code et du mécanisme de livraison [23]. L'intégration continue (CI) et la livraison continue (CD) font partie des pratiques clés de DevOps. Le CI permet aux membres des équipes de développement d'intégrer leur travail fréquemment, généralement plusieurs fois par jour [121, 59]. Le CD vise à garantir que des logiciels de qualité sont produits en cycles courts et qu'ils peuvent être livrés de manière fiable à tout moment [41]. Le CD peut utiliser un ensemble de pratiques, telles que le CI et le déploiement automatisé [71].

Pour tirer pleinement profit de cloud et de DevOps, il est nécessaire de prendre en compte leurs caractéristiques et leurs exigences lors du développement d'applications. Par exemple, dans le cloud, une défaillance peut survenir à tout moment [22, 134]. Par conséquent, les applications à déployer dans cet environnement devraient être en mesure de gérer de telles incertitudes [22]. En outre, pour une adoption plus fluide de DevOps, il est nécessaire d'avoir un système constitué de modules qui peuvent être développés, testés et déployés de manière indépendante. Les systèmes basés sur les microservices peuvent répondre à ces contraintes.

Microservices

Récemment, les microservices [106, 122, 123, 95] sont apparus comme un style architectural bien adapté au développement d'applications à déployer sur le cloud (architecture native pour le cloud [22]), ainsi qu'à celles qui adoptent les pratiques DevOps. En littérature, la définition la plus communément utilisée de ce style architectural est celle proposée par Lewis and Fowler [95]. Selon eux, dans ce style, une application consiste en un ensemble de petits services qui peuvent être déployés indépendamment. Chaque microservice gère ses propres données et communique avec les autres en s'appuyant sur des mécanismes légers. De plus, ils sont couramment emballés et déployés à l'aide de conteneurs [106, 122, 123].

Lors de l'exécution d'une application à base de microservices, ces derniers peuvent être dupliqués indépendamment ou supprimés à la volée pour réduire le temps de réponse ou parce qu'ils ne sont pas utilisés. Une telle duplication dynamique (resp., suppression) nécessite une allocation dynamique (resp., libération) des ressources fournies par le cloud. L'allocation et la libération dynamiques adaptent le coût d'utilisation des ressources aux exigences de l'application. De plus, chaque microservice peut être déployé à l'aide de ressources qui répondent à ses besoins (par exemple, CPU, mémoire, etc.) [52, 114]. Une telle consommation efficace de ressources permet aux fournisseurs de services d'offrir plus de garanties de scalabilité [56]. Enfin, les microservices facilitent la reprise de l'exécution d'une application après une défaillance du cloud, car ils peuvent être redémarrés facilement et rapidement sans qu'il soit nécessaire de redémarrer l'application entière [130].

Outre leur adaptabilité au cloud, les microservices sont bien adaptés au DevOps. Ils permettent une livraison et un déploiement continus d'applications volumineuses et complexes [114]. Cela est dû au fait qu'ils peuvent être développés, testés et déployés séparément. Par conséquent, chaque équipe peut facilement développer, tester et déployer ses microservices indépendamment des autres [130].

Problème et motivation

Ces dernières années, le style architectural microservices est devenu un élément essentiel pour le développement d'applications à déployer sur le cloud ainsi que celles qui adoptent les pratiques DevOps. Néanmoins, de nombreuses applications ont été développées avant l'émergence de ce style en se basant sur un style monolithique.

Problème des Applications Monolithiques dans le Contexte de Cloud et de DevOps

Le style architectural monolithique [128, 52, 114, 95, 112] est la façon traditionnelle de développer des logiciels. Dans ce style, une application, appelée monolithique ou monolithe, est construite comme une seule unité combinant l'interface utilisateur, la logique métier et les accès aux données. Généralement, elle est autonome et indépendante des autres applications.

Le style monolithique est bien adapté au développement de petites applications ou lorsqu'il existe de fortes dépendances entre les différentes parties d'une application, durcissant sa décomposition. Autrement, les monolithes ont des limitations importantes, en particulier lorsqu'ils sont déployés sur le cloud:

— **Surcoût pour l'utilisation des ressources :** la mise à l'échelle d'un monolithe nécessite simplement d'exécuter plusieurs copies de l'application entière derrière un équilibreur de charge [122, 95, 112]. Dans le cloud, les clients sont généralement facturés en fonction des ressources utilisées, même si certaines d'entre elles sont inutilement occupées par des parties du monolithe qui ne sont pas fréquemment utilisées (c'est-à-dire qui implémentent des fonctionnalités moins utilisées) [52]. C'est certainement coûteux de mettre une application à l'échelle de telle façon. En outre, pour déployer un monolithe, les ressources nécessaires sont allouées en fonction des besoins (par exemple, CPU, mémoire, etc.) de l'ensemble du monolithe. S'il existe une variation significative en terme de ces exigences lors de l'exécution du monolithe, les ressources choisies peuvent être sous-optimales ou coûteuses.

— **Temps d'arrêt considérables :** pour reprendre l'exécution après une défaillance du cloud, l'application entière peut être redémarrée. Généralement, le redémarrage d'un monolithe entraîne des temps d'arrêt considérables [52].

— **Difficulté à adopter les pratiques DevOps :** l'adoption des pratiques DevOps n'est pas toujours fluide [146], en particulier lorsqu'il s'agit d'applications monolithiques. L'adoption de DevOps nécessite de disposer d'un système constitué de modules qui peuvent être développés, testés et déployés de manière indépendante. Ce n'est pas le cas lors de la manipulation de monolithes.

En raison de ces limitations, les applications monolithiques ne sont bien adaptées ni au cloud ni au DevOps. Un moyen possible d'adapter ces applications aux deux consiste à les migrer vers un style architectural capable de traiter ces limitations.

Migration des Applications Monolithiques vers des Microservices

Comme expliqué précédemment, les microservices peuvent être développés, testés et déployés indépendamment, ce qui leur permet de remédier aux limitations des applications monolithiques. Pour cela, les monolithes existants peuvent être réécrits à partir de zéro en utilisant des microservices ou migrés vers eux. D'une part, une réécriture complète est considérée comme une tâche risquée, coûteuse et longue. Généralement, la logique métier documentée des monolithes n'est pas à jour [93]. De plus, en raison du roulement du personnel, il devient difficile d'atteindre les experts des applications. En fait, le risque d'échec est généralement trop élevé pour que les entreprises réécrivent leurs applications à partir de zéro [30]. Par ailleurs, la migration vers le style architectural microservices apparaît comme une solution évidente. En effet, il est plus sûr et généralement mené.

Par conséquent, l'objectif ultime de notre thèse est de contribuer à automatiser la migration des applications monolithiques orientées objet vers des microservices afin de les adapter au cloud et au DevOps. Pour cela, nous adressons le problème de recherche qui consiste à répondre à deux questions principales:

1. **Comment identifier, à partir d'une application monolithique orientée objet, l'architecture à base de microservices correspondante tout en respectant autant que possible les principes de ce style architectural?** Cette question est liée à l'observation que certaines approches d'identification de microservices existantes, telles que celles proposées dans [66, 94, 25, 100, 44], présentent des limitations qui les empêchent de répondre en profondeur à cette question. Parmi les principales limitations, nous pouvons citer:

   — Les critères utilisés pour identifier les microservices ne matérialisent pas toute la "sémantique" du concept "microservice". Par exemple, le critère concernant l'autonomie des données des microservices n'est pas toujours pris en compte, en particulier lorsque les entrées requises par l'approche d'identification ne sont pas de la documentation.

   — Pour identifier des microservices à partir d'un monolithe, certaines informations nécessaires peuvent être récupérées en analysant son code source. Outre ces informations, les recommandations fournies par les experts des applications analysés peuvent être exploitées pour améliorer la pertinence/qualité des microservices identifiés. Cependant, les approches existantes qui nécessitent une intervention d'experts souffrent principalement de deux limitations. D'une part, l'intervention des experts ne se limite pas à fournir des recommandations pertinentes, mais ils effectuent plutôt certaines étapes de l'approche. D'autre part, cette intervention est considérée comme toujours nécessaire, ce qui

limite l'applicabilité de ces approches.

— Le modèle algorithmique utilisé a un impact sur la pertinence/qualité de l'architecture à base de microservices identifiée. Les approches existantes reposent principalement sur un nombre limité de modèles tels que le clustering.

Sur la base de ces limitations, la question ci-dessus peut être encore raffinée en trois sous-questions:

— Comment identifier des microservices à partir d'une application orientée objet en tenant compte de la "sémantique" du concept "microservice"?

— Quelles sont les recommandations pertinentes permettant de bénéficier des conseils d'experts et comment peuvent-elles être combinées avec des informations du code source?

— Quel modèle algorithmique peut être utilisé pour identifier les meilleurs microservices possibles en fonction de l'évaluation de leur qualité?

2. **Comment extraire à partir d'une application orientée objet un workflow correspondant utilisé par certaines approches existantes basées sur des tâches pour identifier des microservices?** Cette question est liée au constat que certaines des approches existantes identifient des microservices à partir des workflows [15] ou leurs constituants [44]. Un workflow spécifie le mode de séquencement des tâches suivant deux formalismes: flot de contrôle et/ou flot de données. Un flot de contrôle décrit l'ordre d'exécution des tâches à travers différentes constructions telles que des séquences, des branches conditionnelles (if et switch) et des boucles (for et while). Un flot de données spécifie les échanges de données entre les tâches. Si un workflow n'est pas disponible, une identification basée sur les tâches extrait d'abord celui-ci des artefacts accessibles du monolithe à migrer (par exemple, code source, documentation, etc.), puis l'utilise pour identifier les microservices. Néanmoins, les approches existantes basées sur les tâches supposent soit l'existence de workflows [15], soit les experts peuvent récupérer leurs constituants [44], ce qui n'est pas toujours le cas. Pour garantir l'applicabilité de telles approches, un workflow peut être extrait des artefacts disponibles. Plusieurs approches ont été proposées pour récupérer des workflows. Ce-pendant, elles souffrent de deux limitations principales:

— Généralement, leur objectif n'est pas de récupérer tous les constituants du workflows. Ils extraient soit un flot de contrôle [148, 147, 61, 89, 72, 62, 118, 119, 90] ou un flot de données [43, 33, 99], mais pas les deux.

— Une intervention de l'expert est nécessaire pour effectuer certaines étapes de l'approche manuellement [90], ce qui limite son applicabilité.

Sur la base de ces limitations et de la définition d'un workflow, la question ci-dessus peut être encore raffinée en trois sous-questions:

— Quelles sont les tâches qui reflètent le workflow correspondant à l'application orientée objet?

— Quel est le flot de contrôle à définir entre les tâches identifiées?

— Quel est le flot de données à associer aux tâches et au flot de contrôle identifiés?

Contributions

Pour aborder le problème identifié précédemment, cette thèse présente les contributions suivantes:

1. **Identification des microservices à partir du code source orienté objet basée sur la structure :** notre approche identifie des microservices en se basant sur l'analyse statique du code source orienté objet d'une application monolithique. Elle s'appuie sur des informations liées à la structure du code source, les accès aux données persistantes (c'est-à-dire stockées dans des bases de données) et aux recommandations de l'architecte lorsque ces dernières sont disponibles. Son originalité se décline sous trois aspects:
   — **Identification de microservices basés sur l'évaluation d'une fonction bien définie mesurant leur qualité:** le but de notre approche est d'identifier des microservices en maximisant une fonction qui évalue leur qualité. Elle a été proposée en se basant sur des métriques reflétant la "sémantique" du concept "microservice". Ces métriques ont été définies en raffinant les caractéristiques des microservices. Les paramètres de la fonction de qualité sont les dépendances structurelles du code source et les liens concernant les accès aux données persistantes.
   — **Exploiter les recommandations de l'architecte uniquement quand elles sont disponibles:** un ensemble bien défini de recommandations, qu'un architecte peut fournir, a été spécifié et utilisé dans notre approche. Les recommandations concernent principalement l'utilisation des applications (par exemple, combien de microservices, quelle classe est le centre d'un microservice, etc.). De plus, elles sont combinées avec des informations du code source. Les recommandations, lorsqu'elles sont disponibles, sont impliquées principalement dans le partitionnement des applications orientées objet en microservices. Les informations du code source sont utilisées pour mesurer leur qualité. Selon les recommandations disponibles, différentes identifications de microservices peuvent être effectuées.

— **Modèles algorithmiques de regroupement et de méta-heuristiques pour identifier des microservices en se basant sur l'évaluation de la fonction de qualité:** notre approche regroupe les classes d'une application orientée objet en fonction de leur qualité mesurée à l'aide de la fonction de qualité proposée. À cette fin, deux modèles algorithmiques ont été utilisés: un algorithme de regroupement [142] et un algorithme génétique [69, 103]. La fonction de qualité proposée est considérée comme une mesure de similarité dans l'algorithme de regroupement et comme une fonction objective dans l'autre. Il est possible d'identifier des microservices en utilisant les deux modèles algorithmiques, puis comparer les résultats obtenus et choisir les meilleurs.

2. **Extraction de workflows à partir du code source orienté objet pour permettre l'identification de microservices basée sur des tâches:** pour permettre l'applicabilité de certaines approches d'identification existantes basées sur des tâches [15, 44] lorsque seul le code source est disponible, nous proposons une approche visant à extraire un workflows en analysant le code source. Cette extraction nécessite la capacité de mapper les concepts orientés objet dans ceux du workflow. Par exemple, il est nécessaire de spécifier le mappage du concept tâche par rapport aux concepts orientés objet. Une fois qu'un tel mappage est établi, les constituants du workflow (c'est-à-dire tâches, flot de contrôle et flot de données) sont récupérés à partir du code source orienté objet en s'appuyant sur ce mappage.

Contents

List of Figures

List of Tables

I

Introduction

Over the years, technologies evolve to enable more efficient, and easier software development, deployment, as well as any subsequent activity to software delivery, such as maintenance. Examples of recent technologies allowing that, and attracting the attention of both researchers as well as industrials are cloud computing, DevOps, and microservices.

To keep pace with technological advances, existing applications developed relying on outdated technologies can be rewritten from scratch or migrated to recent ones. Our work contributes to the migration of existing monolithic applications towards microservices to adapt them to both cloud computing and DevOps.

1.1 Context

1.1.1 Cloud Computing and DevOps

Cloud computing is a model for allowing on-demand network access to a shared pool of configurable computing resources such as storage space, servers, applications, services, and so on [101]. Customers can allocate and release these resources quickly with minimum service provider communication and management effort [101]. The cloud is characterized by its high availability, elasticity, scalability, and cost flexibility (i.e., pay-as-you-go model). It provides a variety of services that can be categorized into a hierarchy of *as a Service* terms [127]. A simplified categorization (Figure 1.1) partitions cloud services into three categories: SaaS (Software as a Service), PaaS (Platform as a Service), and IaaS (Infrastructure as a Service). SaaS is a software delivery paradigm, where service providers develop software and deliver it via the Internet [55]. PaaS offers platforms for the development, and deployment of applications in cloud infrastructure using programming languages, libraries, tools, and so on [57]. IaaS providers deliver processing, storage, network, and other fundamental computing resources enabling costumers to deploy and run their software [51, 101]. This can be done by allocating customized Virtual Machines (VMs) with predefined CPU, memory, storage, and bandwidth capacity [127]. Customers can elastically allocate/release VMs, and they are billed based on their usage. Some of the prominent cloud services providers are SalesForce [5], Google App Engine [2], Amazon EC2 [3], and Microsoft Azure [4].

Due to its characteristics, nowadays, the cloud is attracting the attention of both scientific researchers, and industrials. Researchers consider it as a platform enabling them to develop and run massively scalable scientific applications [105], that are usually compute and data intensive. Moreover, instead of upfront investment in an over-provisioned infrastructure, they pay only for what they use. Even companies are attracted to the cloud due to its scalability and cost flexibility [74]. They view it as an opportunistic business strategy allowing them to compete with other companies, and meet business goals [34, 17, 16], especially when merged with efficient DevOps.

DevOps (**Dev**elopment and **Op**eration**s**) [26] is a set of practices aiming to enhance the collaboration between development (e.g., programming, testing, etc.) and operations (e.g., deployment, monitoring, etc.) teams (Figure 1.2) to reduce the time to market. Furthermore, it ensures maintaining the quality of the code and the delivery mechanism [23]. Continuous Integration (CI) and Continuous Delivery (CD) are among the key DevOps practices. CI enables members of development teams to integrate their work frequently, usually multiple times per

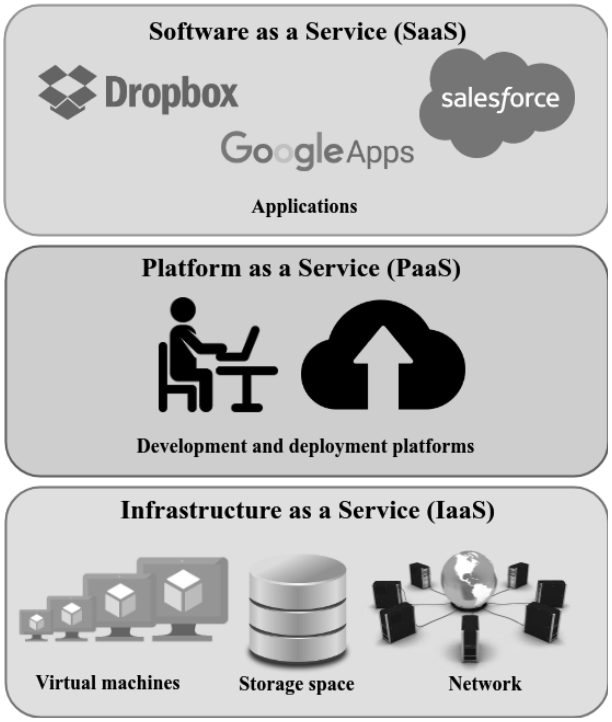

Figure 1.1 – *Cloud computing services*

day [121, 59]. CD aims to ensure that valuable software is produced in short cycles and that it can be reliably released at any moment [41]. CD can use a set of practice, such as CI and automated deployment [71].

Figure 1.2 – *Scheme of DevOps*

Fully benefiting from cloud and DevOps requires taking into account their characteristics and requirements when building applications. For instance, in the cloud, a failure may occur at any moment [22, 134]. Therefore, applications to be deployed in this environment should be able to deal with such uncertainties [22]. Furthermore, smoother adoption of DevOps requires having a system consisting of modules that can be developed, tested, and deployed independently. Microservice-based systems can fulfill these constraints.

1.1.2 Microservices

Recently, microservices [106, 122, 123, 95] have appeared as an architectural style well-adapted for the development of applications to be deployed in the cloud (i.e., a cloud-native architecture [22]), and for those adopting DevOps practices. In literature, the most commonly used definition of this architectural style is the one proposed by Lewis and Fowler [95]. According to them, in this style, an application consists of a set of small services that are independently deployable and scalable. Each microservice manages its own data and communicate with others relying on lightweight mechanisms, generally HTTP resource API (Figure 1.3). Moreover, they are commonly packed and deployed using containers [106, 122, 123].

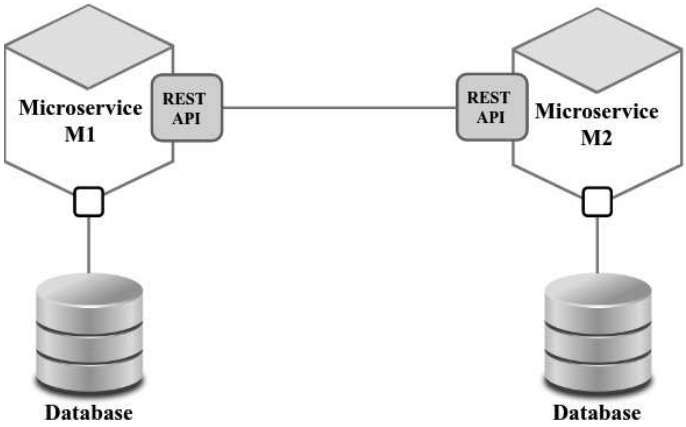

Figure 1.3 – *Representation of microservices*

During the execution of a microservice-based application, microservices can independently be duplicated (i.e., scaled) or deleted on the fly to reduce the response time or because they are not used. Such dynamic duplication (resp., delete) requires a dynamic allocation (resp., release) of resources, which is provided by the cloud due to its elasticity. The dynamic allocation and release adapt the usage cost of resources to the application requirements. Furthermore, each microservice can be deployed using resources that suites its requirements (e.g., CPU, memory, etc.) [52, 114]. Such effective consumption of resources allows cloud service providers to offer higher scalability guarantees [56]. Finally, microservices facilitate resuming the execution of an application after a cloud failure because they can easily and quickly be restarted without the need to restart the entire application [130].

Besides their well-adaptability to the cloud, microservices support DevOps. They enable continuous delivery and deployment of large and complex applica-

tions [114]. This is due to the fact that they can be developed, tested, and deployed separately. Therefore, each team can easily develop, test, and deploy their microservices independently from other teams [130].

1.2 Problem and Motivation

In recent years, the microservice architectural style has become an essential element for the development of applications to be deployed in the cloud and for those adopting DevOps practices. Nevertheless, many applications have been developed before the emergence of this style relying on a monolithic one. Some of these applications, such as the ones used by companies (e.g., Amazon, Netflix, eBay, etc.) afford a high business value.

1.2.1 Monolithic Applications for Cloud and DevOps

Monolithic architectural style [128, 52, 114, 95, 112] is the traditional way to develop software systems. In this style, an application, called monolithic or a monolith, is built as a single unit that combines the user interface, business logic, and data accesses. Usually, it is autonomous and independent from other applications.

The monolithic style is well-adapted for the development of small applications, where a single team works on the code [128], or when there are high dependencies between the different parts of the application hardening the decomposition. Otherwise, monoliths have significant limitations, especially when deployed in the cloud:

— **Extra cost for resource utilization:** scaling a monolith requires simply running multiple copies of the entire application behind a load-balancer [122, 95, 112]. In the cloud, customers are generally billed based on the used resources, even if some of them are unnecessarily occupied by parts of the monolith that are not frequently used (i.e., implementing functionalities less utilized by users) [52]. This is definitely an expensive way to scale an application. Besides this, to deploy a monolith, the needed resources are allocated based on the requirements (e.g., CPU, memory, etc.) of the entire monolith. If there is a significant variation in term of these requirements during the execution of the monolith, the chosen resources can either be sub-optimal or expensive.

— **Considerable downtimes:** to resume the execution after a cloud failure, the entire application may be rebooted. Generally, considerable downtime is entailed when restarting a monolith [52].

— **Difficulty to adopt DevOps practices:** adopting DevOps practices is not always smooth [146], especially when dealing with monolithic applications. The adoption of DevOps requires having a system consisting of modules that can be developed, tested, and deployed independently. This is not the case when handling monoliths.

Due to these limitations, monolithic applications are not well-adapted neither to the cloud nor to DevOps. A possible way to adapt such applications to both is migrating them towards an architectural style that can tackle these limitations.

1.2.2 Migrating Monolithic Applications to Microservices for Cloud and DevOps

As explained earlier, microservices can independently be developed, tested, deployed, and scaled, which allows them to address the mentioned limitations of monolithic applications. To illustrate the limits of these applications and the advantages of microservices, consider the example of Figure 1.4, which shows a monolithic e-commerce application running in the cloud. It takes orders from customers, verifies the availability of the ordered item in the inventory, and ships them. A user can utilize it as a mobile application or via a web browser. Supposing that a high number of new items that may be ordered by users during the upcoming holiday season are being added to the inventory to ensure their availability. Moreover, for the current period, a limited number of orders are being placed. To reduce the response time and guarantee that items are added easily, the entire application should be scaled, even though not all its provided functionalities are frequently used (i.e., placing orders and shipping items). Thus, resources are unnecessarily occupied, and the cost for their usage is paid.

To reduce the response time without paying an extra cost, the monolithic e-commerce application can be migrated to microservices, and only the microservice managing the inventory should be scaled. Figure 1.5 presents a possible decomposition of the e-commerce monolith into microservices (inspired from [113]). The corresponding microservice-based application consists of several microservices, including the *Store Web UI*, which implements the user interface, and three other microservices: *Account microservice*, *Inventory microservice*, and *Shipping microservice*. An API Gateway routes requests from the mobile applications to microservices, which collaborate via APIs.

Since microservices can address the limitations of monolithic applications, existing monoliths can be rewritten from scratch relying on microservices or migrated towards them. On the one hand, a complete rewrite is known as a risky, costly, and time-consuming task. Generally, the documented business logic of monoliths is not up to date [93]. Moreover, due to staff turnover, reaching to applications experts becomes difficult. In fact, the risk of failure is usually too high for companies to rewrite their applications from scratch [30]. On the other hand, the migration towards microservice architectural style appears as an evident solution. Indeed, it is safer and usually carried out.

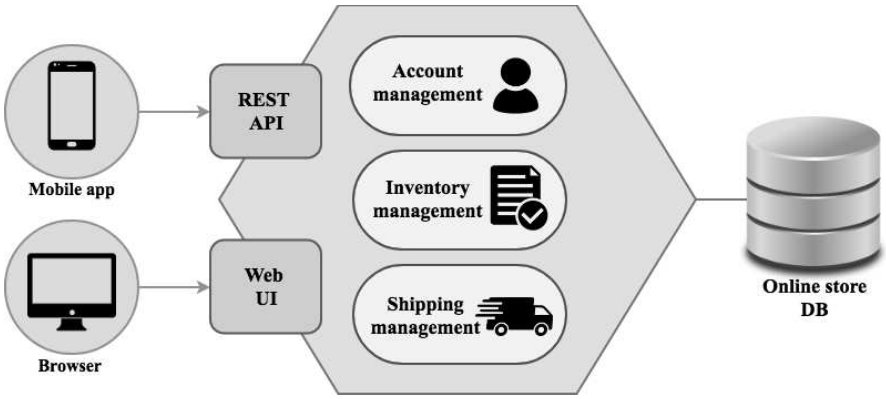

Figure 1.4 – *Example of a monolithic e-commerce application*

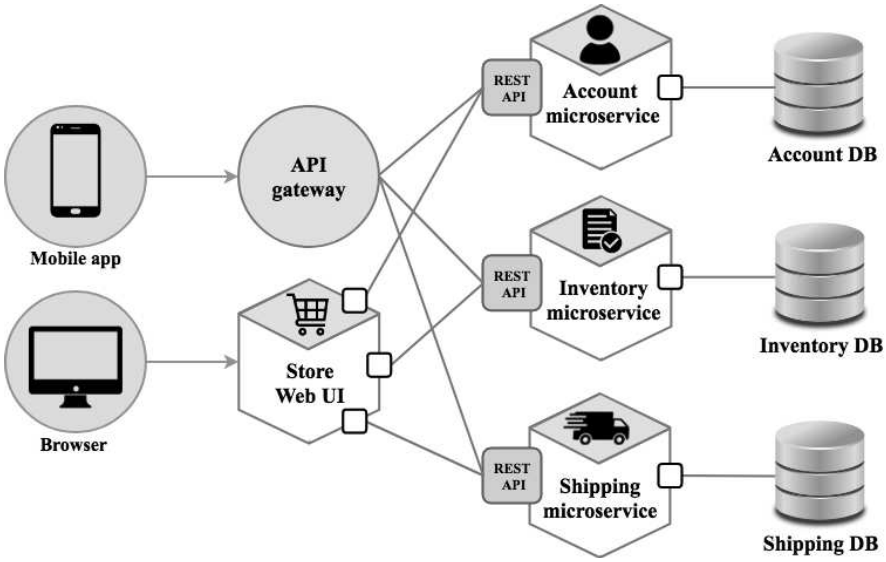

Figure 1.5 – *Example of the monolithic e-commerce application decomposed into microservices*

Therefore, the ultimate aim of our dissertation is contributing to automate the migration of monolithic OO applications towards microservices in order to adapt them to both cloud and DevOps. For that purpose, we address the research problem that consists of answering two main questions:

1. **How to identify from a monolithic OO application the corresponding microservice-based architecture while respecting as much as possible the principles of this architectural style?** This question is related to the observation that existing microservice identification approaches, such as the ones presented in [66, 94, 25, 100, 44], have limitations preventing them from thoroughly answering it. Among the main limitations, we can mention:

   — The used criteria to identify microservices do not materialize all the "semantics" of the concept "microservice". For example, the criterion concerning data autonomy of microservices is not always taken into account, especially when the required inputs by the identification approach are not documentation.

   — To identify microservices from a monolith, the needed pieces of information can be recovered from its source code by analyzing it. Additionally to these pieces of information, the ones (i.e., recommendations) provided by the analyzed application experts can be used to improve the relevance/ quality of the identified microservices. However, existing approaches that require experts intervention suffer primarily from two limitations. On the one hand, the experts intervention is not limited to providing relevant recommendations but rather perform some steps of the approach. On the other hand, experts intervention is considered as always necessary. The aforementioned constraint limits the applicability of these approaches.

   — The used algorithmic model impacts the relevance/quality of the identified microservice-based architecture. Existing approaches rely mainly on a limited number of models such as clustering.

   Based on these limitations, the above question can be further refined into three sub-ones:

   — How to identify microservices from an OO application while taking into account the "semantics" of the concept "microservice"?

   — What are the relevant recommendations allowing to benefit from the guidance of experts, and how can they be combined with source code information?

   — Which algorithmic model can be used to have the best possible microservices based on the evaluation of their quality?

2. **How to extract from an OO application its corresponding workflow to be used by existing task-based approaches to identify microservices?** This question is related to the observation that some existing approaches identify microservices relying on workflows [15] or their constituents [44]. A workflow specifies the sequencing mode of tasks following two formalisms: control flow and/or data flow. A control flow describes the execution order of tasks through different constructs such as sequences, conditional branches (if and switch), and loops (for and while). A data flow specifies data exchanges between tasks. If a workflow is not available, a task-based identification firstly extracts it from the accessible artifacts of the monolith to be migrated (e.g., source code, event logs, etc.), and then use it to identify microservices. Nevertheless, existing task-based approaches either suppose that workflows are available [15] or experts can recover their constituents [44], which is not always the case. To ensure the applicability of such approaches, a workflow can be extracted from the available artifacts. Several approaches have been proposed to recover workflows. However, they suffer from two main limitations:

   — Usually, their aim is not to recover all the constituents of a workflow. They either extract a control flow [148, 147, 61, 89, 72, 62, 118, 119, 90] or a data flow [43, 33, 99], but not both.

   — Expert intervention is necessary to perform some steps of the approach manually [90], which limits its applicability.

   Based on these limitations and the definition of a workflow, the above question can be further refined into three sub-ones:

   — What are the tasks that reflect the workflow corresponding to the OO application?

   — What is the control flow to be defined between the identified tasks?

   — What is the data flow to be associated with the identified tasks and control flow?

1.3 Contributions

To tackle the identified problem in Section 1.2, this dissertation presents the following contributions:

1. **Structure-based identification of microservices from OO source code:** our approach identifies microservices based on the static analysis of OO source code of a monolithic application. It relies on information related to source

code structure, persistent data (i.e., data stored in databases) accesses, and architect recommendations when these latter are available. Its originality can be viewed from three aspects:

- **Identification of microservices based on the evaluation of a well-defined function measuring their quality:** the goal of our approach is to identify microservices by maximizing a function which evaluates their quality. It was proposed based on metrics reflecting the "semantics" of the concept "microservice". These metrics were defined by refining conceptual characteristics of microservices. Parameters of the quality function are source code structural dependencies and accesses links to persistent data.
- **Exploiting architect recommendations only when available:** a well-defined set of recommendations, that an architect can provide, was specified and used in our approach. The recommendations are related, mainly, to the use of the applications (e.g., how many microservices, which class is the center of a microservice, etc.). Moreover, they are combined with source code information. Mainly, the recommendations, when available, are involved in partitioning the OO applications into microservices. Source code information is used to measure their quality. Depending on the available recommendations, different microservice identifications can be carried out.
- **Clustering and meta-heuristics algorithmic models to identify microservices based on the evaluation of the quality function:** our approach group the classes of an OO application based on their quality measured using the proposed quality function. For that purpose, two algorithmic models have been used: clustering [142] and genetic algorithms [69, 103]. The proposed quality function is considered as a similarity measure in the clustering algorithm and as a fitness function in the genetic one. It is possible to identify microservices relying on both algorithmic models, then compare the produced results and chose the best ones.

2. **Workflow extraction from OO source code to enable task-based identification of microservices:** to enable the applicability of some existing task-based identification approaches [15, 44] when only the source code is available, we propose an approach aiming to extract workflows by analyzing the source code. This extraction requires the ability to map OO concepts into workflow ones. For instance, specifying what is the mapping of the concept task compared to the OO concepts is necessary. Once such a mapping is established, workflow constituents (i.e., tasks, control flow, and data flow) are recovered from OO source code relying on it.

Both contributions are positioned within the context of microservice identifi-

cation, as demonstrated in Figure 1.6 (contributions annotated and highlighted in blue boxes). As shown in this figure, both identification approaches (i.e., structure-based and task-based) can have as input source code, and produce as output microservices. The structure-based identification recovers microservices directly, whereas the task-based one, that relies on workflows, firstly extracts them, and then carries out the identification. Moreover, both identifications can be fully automatic or semi-automatic based on whether they use software architect recommendations or not.

1.4 Thesis Organization

The remainder of this dissertation is organized into five chapters as follows:

— *Chapter 2* discusses the state-of-the-art related to migrating monolithic systems towards microservices and extracting workflows from the available software artifacts. In this regards, a taxonomy of the related works is presented to enable comparing and positioning our contributions.

— *Chapter 3* presents our structure-based identification approach that partitions an OO source code into microservices relying on its quality measurement as well as software architect recommendations, when available. Moreover, it uses both clustering and genetic algorithms.

— *Chapter 4* presents our contribution aiming to extract a workflow from an OO application based on static analysis of its source code. The extracted workflow enables a task-based identification of microservices.

— *Chapter 5* presents the conducted experiments to validate our contributions, interprets results, and discusses threats to validity. It starts by validating structure-based microservice identification qualitatively and quantitatively. Then, the workflow extraction is evaluated.

— *Chapter 6* summarizes the work realized in this dissertation and gives some future directions.

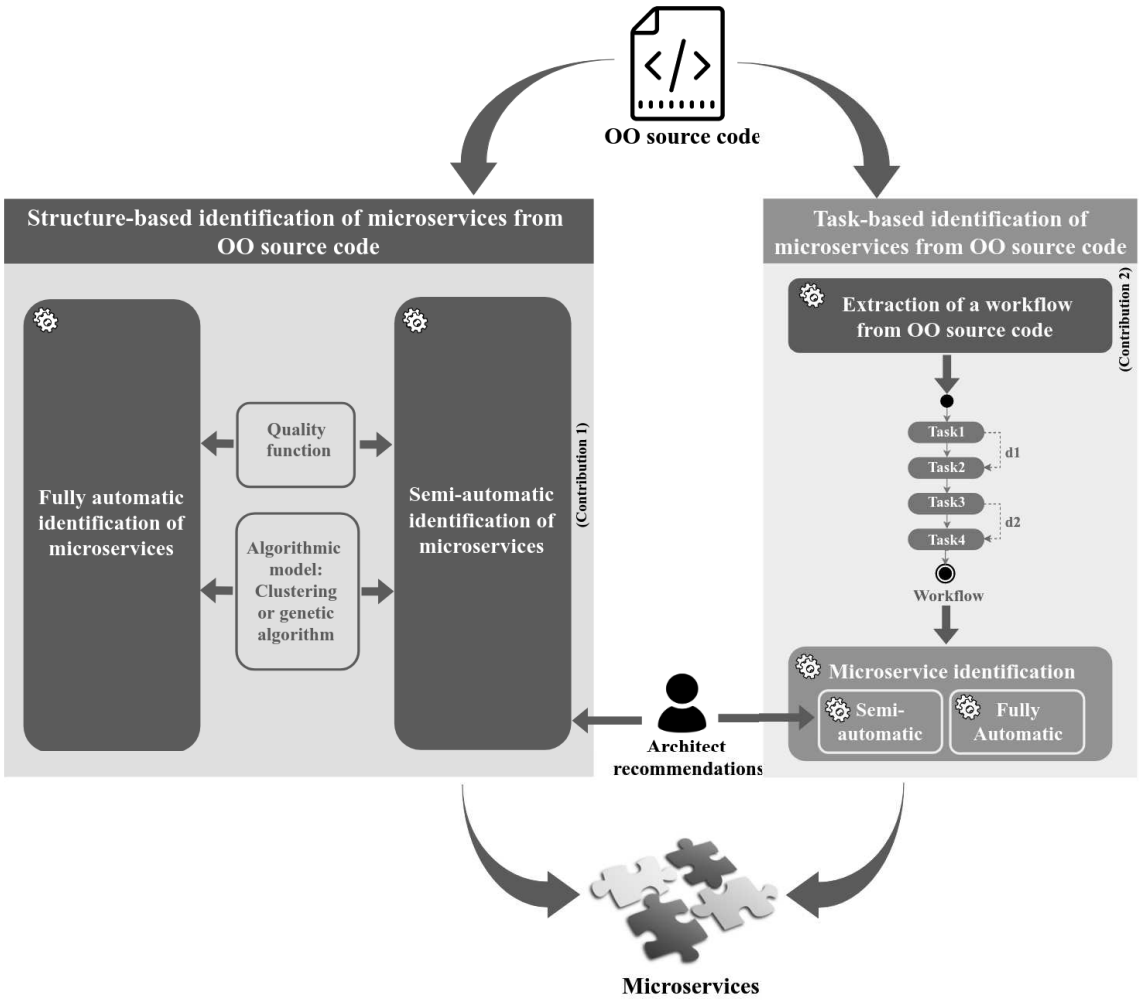

Figure 1.6 – *Positioning the contributions of our dissertation within the context of microservice identification*

II

State of the Art

This chapter discusses the state-of-the-art related to the migration of monolithic applications to microservices. Firstly, Section 2.1 positions our work compared to the relevant domains. Then, Section 2.2 classifies and discusses the related approaches. Finlay, Section 2.3 concludes this chapter.

2.1 Brief Background on Migrating Monolithic OO Software Towards More Recent Technologies

2.1.1 Software Reengineering and Migration

Software reengineering is defined as an engineering process seeking to generate an evolvable system [120]. Generally, it includes all the subsequent activities to software delivery that aims at improving the understanding of the software as well as enhancing various quality parameters, such as system maintainability and complexity [137]. It is defined by Chikofsky and Cross II [45] as:

"Software reengineering, also known as both renovation and reclamation, is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

Based on this definition, the software reengineering process consists of three steps [137]. Firstly, the subject system is examined to recover relevant information (i.e., reverse engineering). The second step alters the system (i.e., transformation). Finally, the new implementation is generated (i.e., forward engineering).

When the reengineering is driven by a major technology change, it can be called migration [10], for instance, the migration of OO systems to component-based ones. In other words, software migration is a variant of software reengineering [10]. It is defined by Bisbal et al. [29] as:

" Migration (...) allows legacy systems to be moved to new environments that allow information systems to be easily maintained and adapted to new business requirements, while retaining functionality and data of the original legacy systems without having to completely redevelop them."

The reengineering/migration process can be represented by a horseshoe model (Figure 2.1) [84], that displays the steps mentioned above. This model includes three levels of abstraction: code representation, function representation, and architecture representation.

1. **Reverse engineering:** reverse engineering is the process of recovering high-level abstractions, such as software architecture, by analyzing lower level ones, like source code [45]. Its outcome is the basis for the next steps (i.e., transformation and forward engineering). For instance, when migrating an OO application to microservices, reverse engineering can be applied to recover the architecture of the existing application (i.e., its classes and their relationships). In literature, many terminologies referring to reverse engi-

neering have been used, among them extraction [72], mining [62], identification [77] and recovery [148]. In our dissertation, these terms are used to refer to reverse engineering.

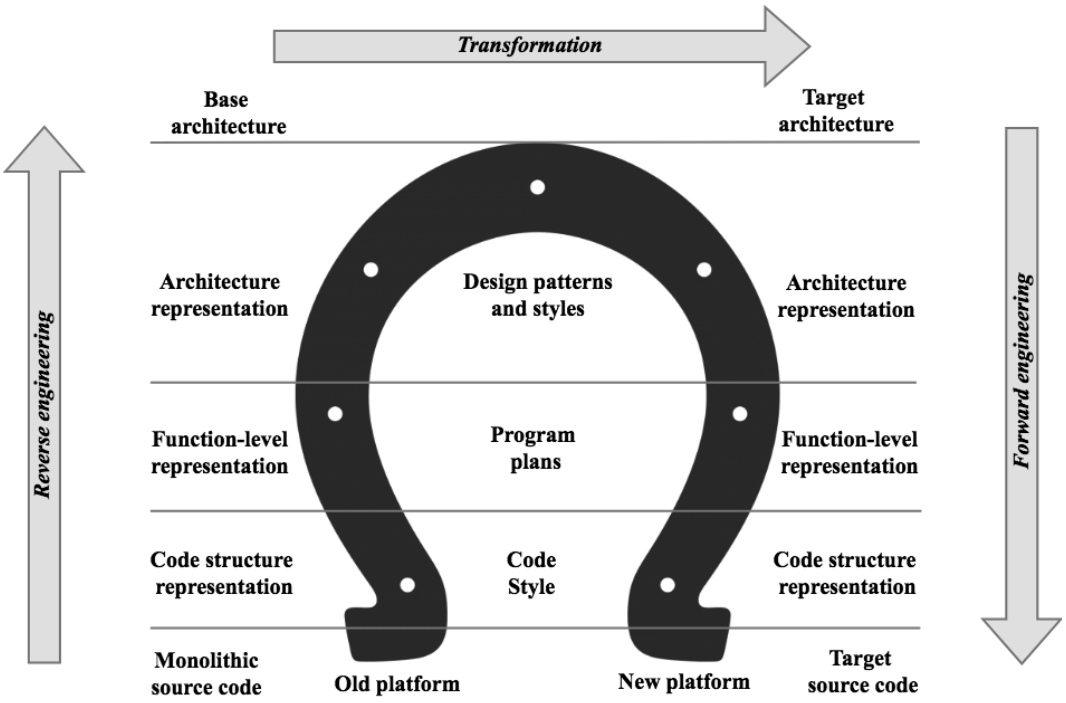

Figure 2.1 – *Horseshoe model representing the reengineering/migration process*

2. **Transformation:** also known as restructuring, it refers to transforming one representation form to another at the same level of abstraction (e.g., source-to-source, model-to-model, etc.) while preserving the functionalities of the system [45]. In [84], it is called architectural transformation. It aims to transform the reverse engineered architecture to the desired one. For instance, in this step, the recovered architecture of a monolithic OO application can be transformed into a microservice-based one. This may be done by partitioning the classes of the OO architecture based on their relationships. Each cluster represents a microservice. It is noteworthy that the transformation is not included in the reverse engineering [10]. Nevertheless, in literature and our dissertation, the two steps have been referred to by reverse engineering.

   During the transformation, the structure of a system can be improved. This is known as refactoring. It is defined in [58] as: *"the process of changing a software system in a way that does not alter the external behavior of the code yet improves its structure. It is a disciplined way to clean the code that minimizes the*

chances of introducing bugs. In essence, when you refactor, you are improving the design of the code after it has been written." The refactoring enables reworking a bad code into well-structured one by applying simple changes, such as pulling some code out of a method [58]. The cumulative effect of these simple changes can thoroughly enhance the structure of the system [58].

3. **Forward engineering:** forward engineering is the traditional process of moving from high-level conceptual abstractions to the physical implementation of a system [45]. For example, when migrating an OO application to microservices, forward engineering can be applied to generate an implementation of the microservice-based application relying on the recovered architecture (i.e., identified microservices) in the previous step.

2.1.2 From Object-Oriented to Components, Services and Microservices

2.1.2.1 Evolution of Technologies

Over the years, technologies evolve to ease the development of applications, their deployment, as well as any subsequent activity to their delivery. Figure 2.2 shows the evolution of technologies from Object-Oriented Programming (OOP) to microservices.

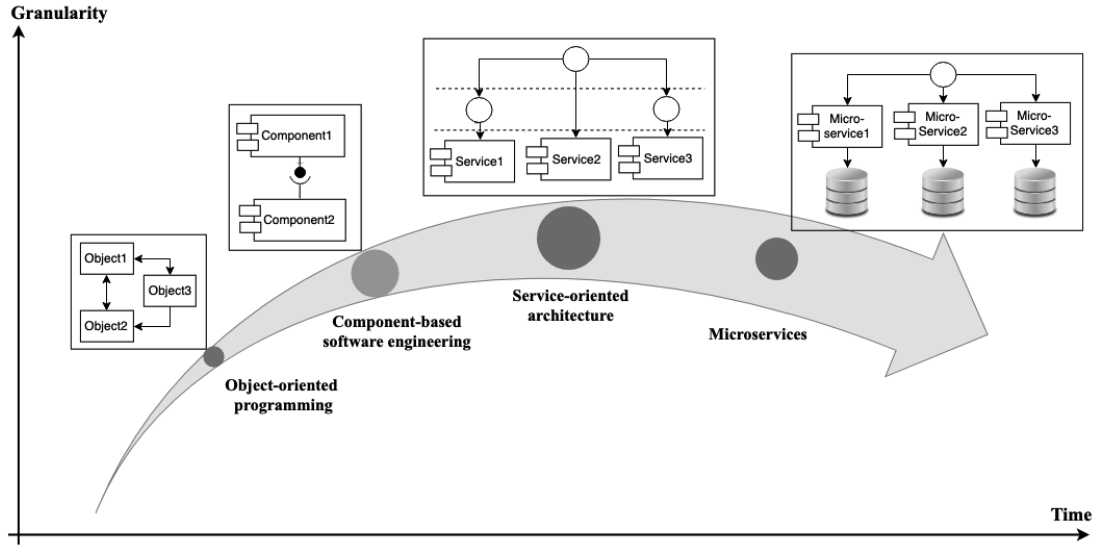

Figure 2.2 – *Evolution of technologies*

In the 1970s, OOP emerged as a new paradigm in which an application mainly

consists of a set of objects that interact with each other. It became broadly used in the 1980s and early 1990s [7] due to its features. The principal ones are object composition, class inheritance, abstraction, encapsulation, and polymorphism [47]. However, the main problem with OOP is that OO classes can have complex mutual dependencies hardening their effective reuse [140] as well as deployment, complicating maintenance, and increasing costs [12]. Several technologies that can tackle this problem have appeared: components, services, and microservices.

Component-Based Software Engineering (CBSE) aims to build applications by selecting off-the-shelf components, and then assemble them [35]. Each component can be seen as a black box, whose implementation is entirely hidden behind well-specified interfaces [7]. In fact, components interact with each other via their required and provided interfaces that explicitly specify their dependencies, and offered functionalities [124]. Since an application is built as a composite of subparts, it enables increased reuse and reduced production cost [91].

Service-Oriented Architecture (SOA) *"is an approach that addresses the requirements of loosely coupled, standards-based, and protocol independent distributed computing"* [108]. It enables building an application that provides services either to end-user applications or to other services distributed in a network, relying on published and discoverable interfaces [107]. SOA can be considered as an evolution of CBSE [8]. Potentially, for this reason, there are several similarities between them. Both service-based applications and component-based ones consist of interconnected services and components respectively. Moreover, both services and components are autonomous entities that enable using their functionalities only via their interfaces [8]. Nevertheless, one of the main characteristics of services is that they are technology and platform independent, which is not always the case for components [81].

Microservices [1] [106, 122, 123, 95] can be viewed as an architectural style in which an application consists of a set of small services that are independently deployable and scalable [95]. Each microservice manages its own data [95, 106] and communicate with others relying on lightweight mechanisms, generally HTTP resource API [95]. Furthermore, they are commonly packed and deployed using containers [106, 122, 123]. At a very high level, there are some similarities between SOA and microservices [114]. Both of them build an application as a set of services communicating with each other. Nevertheless, digging deep reveals significant differences. Firstly, in SOA, services are coarse-grained, whereas in microservice architectural style, even though services may not always be fine-grained, they are smaller [114], which make their development and maintenance easier. Moreover, the bigger the services are, the less they become reusable [7].

1. In literature, the most commonly used definition of microservices is the one proposed by Lewis and Fowler [95].

Therefore, microservices can be more reusable than services. Secondly, unlike services in SOA, each microservice typically has its own database. Thus, it can be more autonomous. Finally, SOA and microservices rely on different technology stacks [75, 114]. Usually, SOA-based applications use heavyweight technologies (e.g., SOAP, WS* standards, etc.), whereas microservice-based applications tend to utilize lightweight ones (e.g., REST, gRPC, etc.) [75, 114].

2.1.2.2 Migration of Monolithic Object-Oriented Applications to Components, Services, and Microservices

Besides the adoption of components, services, and microservices for the development of new applications, migrating existing monolithic OO ones towards these technologies can allow them to benefit from their advantages.

On the one hand, migrating OO applications to component-based ones, commonly, consists of three main steps [12]. Firstly, components are identified from OO source code or any other available software artifacts, such as documentation. Secondly, the required and provided interfaces of each component are identified to enable assembling them. Finally, the OO application is transformed into an operational component-based one relying on the constructed architecture (i.e., components and their interfaces). Several approaches have been proposed to contribute to the migration of OO applications to component-based ones [140, 38, 28, 53, 11, 12, 85, 13].

On the other hand, the migration of OO applications to SOA consists, commonly, of two major steps [7, 125]. The first one identifies services from the available artifacts, whereas the second step packages and deploys them. Usually, it wraps the identified services by interfaces and orchestrates their operations [7]. Several approaches have been proposed to migrate OO applications towards SOA [144, 39, 145, 126, 40, 8, 125, 18].

Nowadays, OO applications are being migrated towards microservices. Similarly to SOA, the migration process consists mainly of two steps: 1) microservice identification and 2) microservice packaging and deployment, typically using containers such as Docker. In our work, we focus on microservice identification, which is a complex [100] and crucial step for a successful migration. Misidentifying microservices limit benefiting from their advantages. Furthermore, it impacts negatively the quality of service (e.g., response time, etc.).

2.2 Taxonomy of Related Works

In the context of microservice identification from monolithic applications, a considerable number of relevant approaches have been proposed. In these approaches, either a structure-based or a task-based identification has been carried out. The structure-based identification directly manipulates the required inputs (e.g., source code, change history, etc.) to partition a monolith into microservices. Whereas, the task-based one firstly recovers a workflow or its constituents (i.e., tasks, control flow, or data flow), if they are not available, and then uses them to decompose the monolith into microservices. Therefore, workflow extraction approaches are considered as related works. Note that by workflow extraction, we refer to recovering the entire workflow or its constituents. This section aims to investigate, classify, and discuss the related works.

2.2.1 Technical versus Feedback Approaches

Several approaches have been proposed to identify microservices or extract workflows. Preliminarily, these approaches can be classified into two categories:

1. ***Technical approaches:*** it includes all the systematic approaches aiming to identify microservice [66, 94, 86, 25, 100, 44, 15, 76, 77] or to extract workflows [43, 33, 99, 148, 61, 147, 89, 72, 62, 118, 90, 119]. They have well-defined steps specifying how to produce the desired outputs using the required inputs.

2. ***Feedback approaches:*** this category includes the approaches presenting experiences, feedbacks, and lessons learned from migrating monolithic applications to microservices or extracting workflows. To the best of our knowledge, no such approaches have been proposed for workflow extraction. The existing ones, included in this category, concern migrating monoliths to microservices [23, 65, 32, 87, 31, 64, 60, 42].

Since our proposed approaches are included in the first category, our focus will be on its approaches. Therefore, the remainder of this section, firstly, investigates, classifies, and discusses the related technical approaches, starting with the ones identifying microservices, followed by the ones extracting workflows. Then, feedbacks and learned lessons from migrating monolithic applications to microservices are presented and discussed.

2.2.2 Technical Approaches

2.2.2.1 Microservice Identification Approaches

As mentioned earlier, a microservice identification approach can be structure-based or task-based. Moreover, its life cycle is composed of objective, inputs, process, and outputs [124]. Therefore, this section, firstly, classifies the identification approaches relying on whether they are structure-based or task-based. Secondly, they are assorted based on four main dimensions (i.e., multi-dimensional taxonomy): the targeted objective, the required inputs, the applied process, and the desired outputs (Figure 2.3). Finally, the taxonomy results are discussed and the key findings to keep in mind are presented.

Structure-based versus Task-based Taxonomy

Based on what do they consider when identifying microservices from monolithic applications, there are two main categories of identification approaches:

— **Structure-based identification:** the structure-based identification considers microservices as separate entities, which can encapsulate operations and communicate with each other. Typically, each entity is focused on one functionality. To partition a monolith into microservices, structure-based identification employs the structural relationships between its entities. Here, structural relationship refers to any relationship in which the temporal evolution of the execution of the monolith is not taken into account (e.g., semantic similarity, data manipulations, etc.).

  For example, in [100], the authors have proposed an approach to identify microservices from the source code of a monolithic application. They have used three coupling strategies (i.e., logical, contributor, and semantic coupling) and embed those in a graph-based clustering algorithm. The logical coupling strategy considers that the classes changed together should be in the same microservice. The contributor coupling strategy assumes that the classes modified by the same contributors are more coupled. Semantic coupling strategy considers that the classes containing code about the same "thing" should be in the same microservice. None of these strategies take into account the temporal evolution of execution while identifying microservices, thus they are structure-based. In fact, most of the investigated approaches [66, 94, 86, 25, 100, 76, 77] are structure-based.

— **Task-based identification:** the task-based identification considers microservices as a set of highly inter-connected tasks. Potentially, each set represents a well-specified functionality. The connection can be determined based on the execution order of these tasks, and their manipulated data (i.e., read

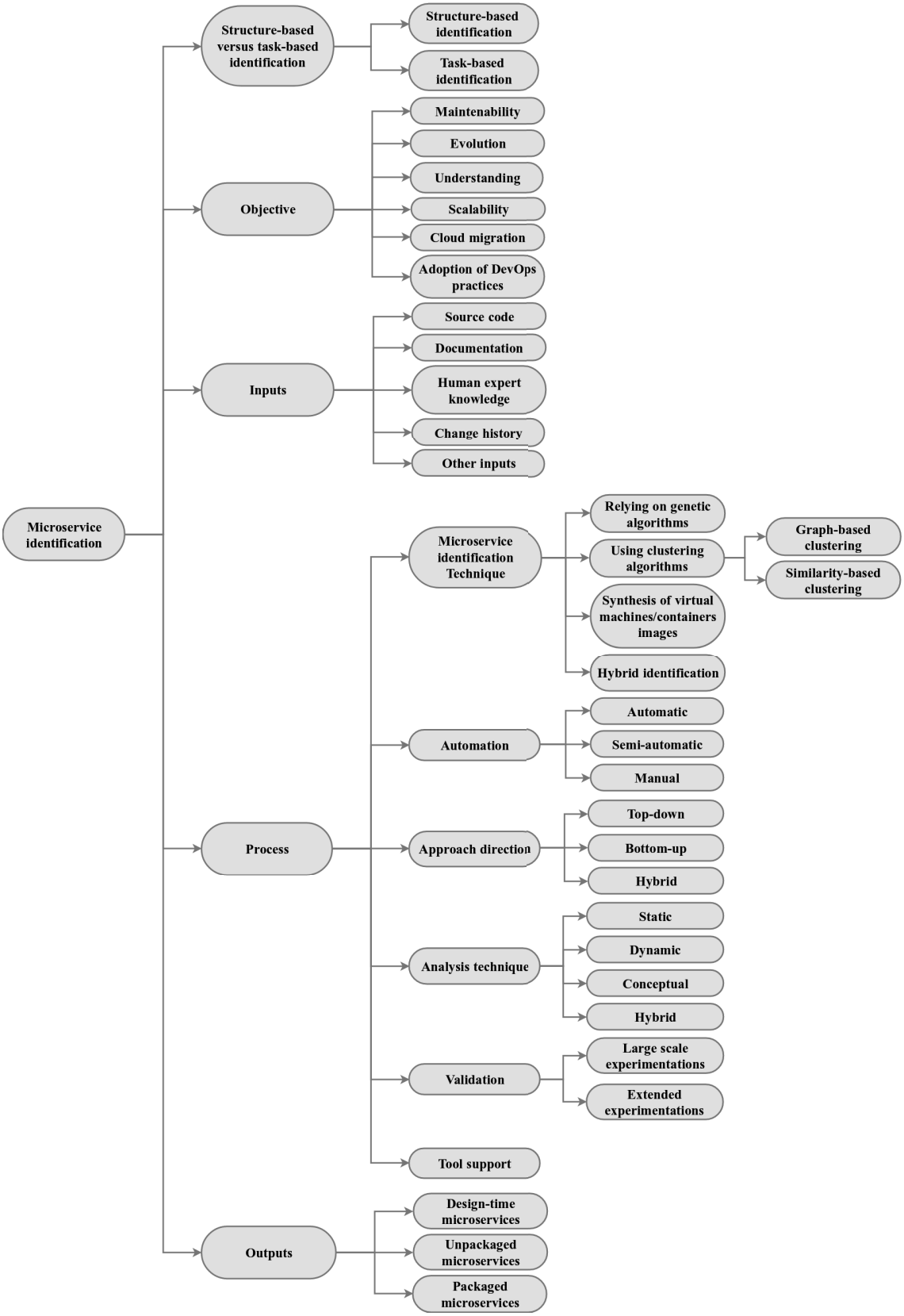

Figure 2.3 – *Taxonomy scheme of microservice identification approaches*

and/or written data). The ones executed sequentially and manipulating the same data are more likely to be in the same microservice.

For instance, in [15], the documented workflows (i.e., business processes) has been partitioned into microservices based on the execution order of tasks, measured by the relation T_P, and their manipulated data, evaluated by the relation T_D. Among the investigated approaches, only two [44, 15] are task-based.

To better understand the structure-based and task-based identifications, Figure 2.4 shows an example of the produced microservices by the two types of identification. On the left side, the structure-based one decomposes the classes of an OO source code, for instance, based on their semantic similarity, into two microservices. Each one consists of a set of classes. On the right side, the task-based identification also produces two microservices. Each one of them consists of a set of tasks executed sequentially. For example, the execution of *Task 6* precedes that of *Task 5*. Moreover, the tasks of each microservice manipulate the same data. For instance, *Task 5* defines *Data 4* and *Data 5* used by *Task 6*.

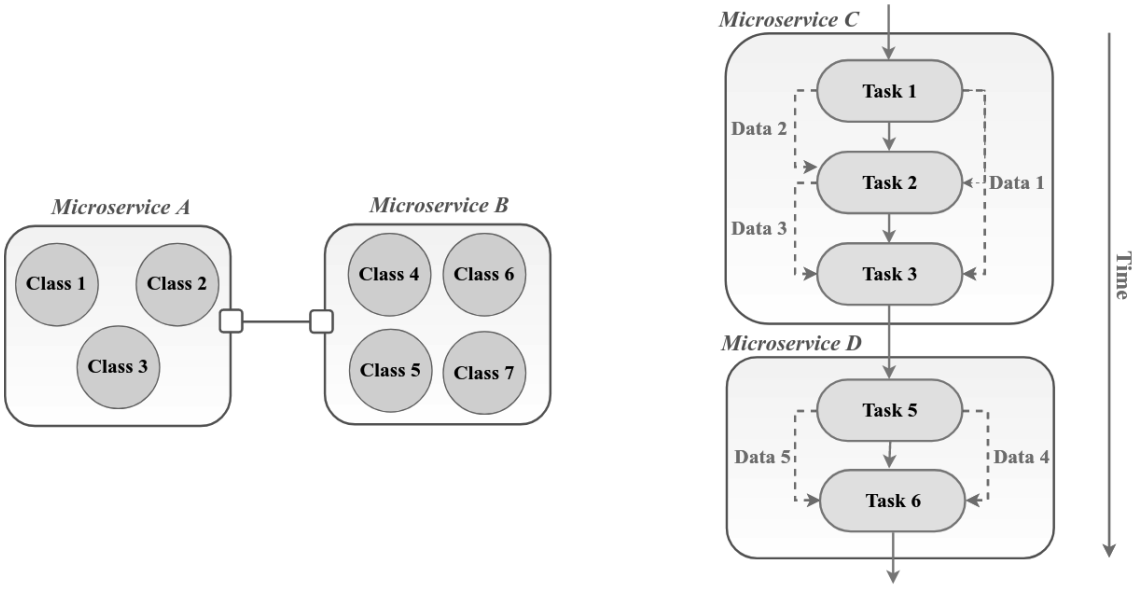

Figure 2.4 – *Example of structure-based and task-based identifications of microservices*

Now that structure-based and task-based identifications have been presented, the question is which identification to use when migrating a monolithic application towards microservices? To answer this question, the two types of iden-

tification will be compared in the remainder of this section. The comparison is based on the main criteria that can assist and facilitate the choice. These criteria are the following: objective, partitioned artifacts into microservices, and resource utilization.

Firstly, both types of identification have the same objective. It is producing microservices that materialize the semantics of the concept microservice. In fact, the generated results by these two types of identification have the main characteristics of microservices, at least some of them, such as providing a well-specified functionality.

Secondly, structure-based and task-based identifications partition different artifacts into microservices. On the one hand, the structure-based identification decomposes artifacts that define the entities of the monolith and their structural relations. An example of such artifacts is OO source code. Analyzing it enables recovering its classes and their relationships (e.g., methods calls, inheritance, etc.). On the other hand, the task-based identification partitions artifacts that express the execution order of the different tasks as well as their exchanged data, such as workflows. Note that a task-based identification recovers these artifacts if they are not available, and then use them to identify microservices.

Thirdly, unlike structure-based identification, the task-based one potentially produces microservices that can be executed following a well-specified order. This is due to the fact that they have been identified relying on the execution order of tasks. Knowing when each microservice can be executed enables more efficient use of resources, especially in the cloud, where customers are charged based on their usage (i.e., pay-as-you-go model). The resources can be allocated when needed to deploy and run a microservice and released when its execution is over.

Relying on this comparison, carrying out a structure-based or task-based identification depends mainly on 1) the available artifacts to partition into microservices and whether it is possible to extract, if not accessible, from them the ones expressing the execution order of the monolith or not, and on 2) the importance of optimizing resource usage. In some cases, all the microservices need to be executed in the cloud permanently because they can be used at any time. For instance, the microservices of an e-commerce application can be utilized by a customer to order new items at any moment. Here, optimizing resource utilization is not a principal concern. However, when running a scientific application to produce results, optimizing resource usage can be a primary concern, especially that these applications are compute and data intensive.

Multi-dimensional Taxonomy

As explained earlier, microservice identification approaches can be classified based on four main dimensions: the targeted objective, the required inputs, the applied process, and the desired outputs.

— **Objective of microservice identification approaches:** the objectives of the investigated approaches can be one or many among the following ones:

1. **Maintainability:** maintainability is defined by ISO/IEC 25010:2011 [6] as *"the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers."* The modifications may be performed with the aim of 1) correcting, 2) improving or 3) adapting an application to changes in environments, requirements, and functional specifications. Decomposing a monolithic OO application into microservices can enhance its maintainability, mainly because the small size of each microservice contributes to increasing code understandability [130] and limits the scope of bugs [52].
2. **Evolution:** according to Lehman [92], software evolution is an intrinsic and feedback-driven process. It refers to developing an application and continuously changing it during its life cycle for different reasons (e.g., organizational and environmental feedbacks, etc.) [92]. Similarly to maintainability, the small size of microservices can facilitate their evolution. For this reason, some approaches, such as the one presented in [94], identify microservices from monolithic applications.
3. **Understanding:** before maintaining or evolving an application, it should firstly be understood [136]. When dealing with monoliths, usually developers, particularly new ones in development teams, are intimidated by the large monolithic code base [112], because it can not be understood easily. Modern development technologies, such as microservices, can facilitate understanding applications. An application developed using these technologies consist of modules. Therefore, instead of understanding the entire application, only the concerned modules are considered.
4. **Scalability:** scalability was defined in [78] as: *"scalability means not just the ability to operate, but to operate efficiently and with adequate quality of service, over the given range of configurations. Increased capacity should be in proportion to the cost, and quality of service should be maintained."* Thus, scalability can be considered as the ability of an application to process more workload while maintaining the same quality of service and taking into account the cost. This can be achieved by using more resources (i.e., their number increases) or more powerful ones (i.e., constant number of resources) [141]. In paying platforms, such as

the cloud, customers are usually charged based on the used resources. Therefore, adjusting this usage, by scaling only the modules of the application having a high workload, enables reducing the cost. To enhance their scalability, monolithic applications have been partitioned into microservices in [77].

5. **Migration to the cloud:** microservices are well-adapted to the cloud. They can facilitate the reconfiguration of an application according to the changes that may occur during its execution. These changes can be related to cloud resources (e.g., resource allocation, release, etc.), quality of service (e.g., reduce response time, etc.) or any other event (e.g., unexpected failure, etc.). Therefore, to migrate monolithic applications to the cloud, some existing approaches decompose them into microservices [44, 100, 15].
6. **Adoption of DevOps practices:** efficient DevOps merged with efficient use of cloud resources produce high Information Technology (IT) performance [80], which is the aim of today's highly competitive companies. Nevertheless, adopting DevOps practices is not always smooth [146], especially when an application is monolithic. Smoother adoption of DevOps requires having a system consisting of modules that can be developed, tested, and deployed independently. Microservice-based systems can fulfill this constraint. For this reason and similarly to our approach, some existing works [100] partition monoliths into microservices.

Table 2.1 presents the classification results of the investigated approaches based on their objective. It is noteworthy that some approaches [44, 15] aims to tackle the limitations of monolithic applications, without specifying which one exactly. Therefore, their objective is considered as a combination of all the objectives related to overcoming these limitations and targeted by the investigated approaches: maintainability, understanding, evolution, scalability, cloud migration, and adoption of DevOps practices.

— **Inputs of microservice identification approaches:** the required inputs by identification approaches can be source code, documentation, human expert knowledge, and change history.

1. **Source code:** source code is a set of computer instructions enabling to describe the execution of a software system [67]. Developers commonly write these instructions using computer programming languages such as Java, C, C++, and so on [124]. In the case of OO programming, the source code primarily consists of a set of classes organized using packages [115]. The main units of each class are methods and attributes.

Table 2.1 – *Classification of the investigated microservice identification approaches based on their objective*

Approach	Maintainability	Evolution	Under-standing	Scalability	Migration to the cloud	Adoption of DevOps practices
Gysel et al. [66]	✓			✓		
Lev-covitz et al. [94]		✓				
Kecskemeti et al. [86]				✓		
Chen et al. [44]	✓	✓	✓	✓	✓	✓
Mazlami et al. [100]					✓	
Baresi et al. [25]				✓		
Amiri [15]	✓	✓	✓	✓	✓	✓
Jin et al. [76]		✓				
Jin et al. [77]	✓			✓		

Due to its availability, some of the investigated approaches [94, 100] use the source code. They partition its classes into disjoint clusters based on the relationships between them. In the investigated approaches, these relationships are mainly of three types: structural (e.g., method invocations, attribute accesses, etc.) [94], semantical [100] or related to data manipulations [94]. For instance, in [100], the content and semantics of class files are examined through information retrial techniques to group the classes containing code about the same thing into the same microservice.

2. **Documentation:** any text, model, or illustration associated with a software system aiming to explain how it operates or how can it be used is considered as documentation [110]. It can be requirement documentation, such as use cases, design documentation, like software architecture [10], and so on. Moreover, it can express data manipulations and exchanges between the entities of a system. For instance, in [15], the required input is a business process (i.e., documented workflow) in which the manipulated data by each task are specified. Whereas, in [44], the input is a text describing the business logic of a monolithic

system to be decomposed into microservices. This text is used to build a data flow specifying data dependencies between different tasks.

3. **Human expert knowledge:** human experts are a source of relevant information regarding the design and the implementation of a software system [124]. Among the investigated works, only two [66, 44] requires experts interventions, not to provide recommendations but rather perform some steps manually. In [44], engineers and users conduct business requirement analysis to build a Data Flow Diagram (DFD) [143]. The constructed diagram is then used to identify microservices. In [66], human interaction is needed to provide a machine-readable representation (i.e., using JavaScript Object Notation (JSON)) of the inputs (e.g., Entity-Relationship Models (ERMs), use cases, etc.).
4. **Change history:** the change history of the source code enables tracking the performed changes on a system. In the investigated approaches, it was only used by the approach presented in [100] to identify microservices relying on logical or contributor coupling strategies.
5. **Other inputs:** some approaches require other inputs. For instance, in [76, 77], the executable system (i.e., executable instance built from the targeted monolithic software) as well as pre-prepared test cases (i.e., test suite) covering as many functionalities as possible are needed. In [86], the input is monolithic services deployed in cloud VMs. According to Kecskemeti et al. [86], services *"are mostly delivered as a monolithic block of multitude of sometimes vaguely related functionalities"*, even though this might not be conform with the "semantics" of the concept "service".
6. **Combination of multiple inputs:** some approaches use a combination of the presented inputs. It can include inputs of the same type (e.g., documentation, etc.) or different ones. Moreover, in some cases, the combination is mandatory, while in others, it is optional. An example of an approach combining optional inputs of different types is presented in [100]. In this approach, depending on the chosen coupling strategy, the input is either the source code or the change history. In [44], two inputs of different types (i.e., documentation and expert knowledge) are both required.

Table 2.2 presents the classification results of the related microservice identification approaches based on the required software artifacts as inputs.

— **Process of microservice identification approaches:** the process of an identification approach refers to the way this approach achieves its objective using the required inputs, and how it is validated. This process is composed

Table 2.2 – *Classification of the investigated microservice identification approaches based on their required inputs*

Approach	Source code	Documentation		Human expert	Change history	Other inputs
		Type	Expressing data manip. and exchages			
Gysel et al. [66]		System specification artifacts (e.g., ERMs, use cases, etc.)	✓	✓		
Levcovitz et al. [94]	- Structural relationships - Data manipulations relationships			✓		
Kecskemeti et al. [86]				✓		Monolithic services deployed in cloud VMs
Chen et al. [44]		Text	✓	✓		
Mazlami et al. [100]	Semantical relationships				✓	
Baresi et al. [25]		OpenAPI specifications				
Amiri [15]		Business process	✓			
Jin et al. [76]						- Executable system - Test cases
Jin et al. [77]						- Executable software - Test cases

of six aspects: 1) microservice identification technique, 2) automation 3) direction, 4) analysis technique, 5) validation, and 6) tool support.

1. **Microservice identification technique:** the microservice identification technique refers to the way the process manipulates the required inputs to produce the desired outputs. There are four main techniques:
   (a) **Relying on genetic algorithms:** genetic algorithms [69, 103] are meta-heuristics typically used when the search space is large, and a fitness function can evaluate a candidate solution (e.g., a possible

decomposition of a monolith into microservices, etc.). The basic idea of genetic algorithms is to start from a set of initial solutions and use evolutionary mechanisms inspired by biology (e.g., selection, mutation, etc.) to create new and potentially better solutions. For instance, the presented approach in [15] uses a genetic algorithm to partition the tasks of a business process [2] into microservices. The possible partitions were evaluated by the turbo-MQ fitness function [102] that takes into account the execution order of tasks, as well as their manipulated data.

(b) **Using clustering algorithms:** clustering algorithms [142] aims to partition a set of entities into clusters (i.e., groups, subsets, or categories) so that the entities of a cluster are more similar to each other than to the ones belonging to other clusters. The used clustering algorithms in the investigated approaches can be classified into two categories:

i. **Graph-based clustering:** this clustering consists of building a graph from the available inputs and then use it to identify microservices. The constructed graph can be partitioned into microservices or they are extracted relying on the dependencies between its nodes.

An example of the first case is presented in [100], where a graph is built from the source code. In this graph, the nodes correspond to the classes of the monolithic application. Each edge has a weight evaluated by a weight function that determines how strong the coupling between two nodes is based on the chosen coupling strategy (i.e., logical, contributor, and semantic coupling). In this approach, the proposed weight functions do not consider all the "semantics" of the concept "microservice". For instance, none of them take into account data autonomy, even though it is one of the main characteristics of microservices.

An example of the second case is presented in [94], where microservices are identified from monoliths relying on dependencies between facades and databases tables connected by business functions. The identification is based mainly on the paths between the nodes of the constructed graph. The nodes correspond to facades, business functionalities, and database tables. Facades represent the entry points of the system. They invoke business functionalities. Edges are created between nodes if there are 1) calls from facades to business functionalities, 2)

2. A workflow is the automation of a business process [70]. Thus, the later can be considered as a documented workflow.

calls between business functionalities or 3) accesses from business functionalities to database tables. This approach takes into account the data autonomy of microservices when identifying them.

ii. **Similarity-based clustering:** this clustering identifies microservices directly relying on the similarity between the available inputs. For instance, the proposed approach in [25] decomposes a monolith into microservices based on the semantic similarity between its operations described in OpenAPI specifications [3].

(c) **Synthesis of virtual machines/containers images:** in [86], to identify microservices from monolithic services, the authors rely on the ENTICE environment. This environment enables VM and container images management (e.g., creation, assembly, storage, etc.). The idea is once the images are created for the monolithic service, the microservice developer can use the ENTICE environment to create customized VM/container images that focus only on the desired functionality of the microservice. The main ENTICE environment techniques used to identify microservices are image synthesis and image analysis.

(d) **Hybrid identification:** any combination of the previously presented techniques is considered as a hybrid one. Among the investigated approaches, only one [77] relies on both similarity based clustering and genetic algorithms. In this approach, the execution traces are firstly analyzed to generate clusters of classes, called functional atoms. Each of them represents a minimal coherent unit, in which the classes are responsible for the same functional logic. The similarity between atoms is measured using the Jaccard Coefficient. Once the functional atoms are generated, they are grouped relying on a tailored Non-dominated Sorting Genetic Algorithm-II (NSGA-II) [50]. The classes of each group correspond to a candidate microservice.

Table 2.3 presents the classification results of the investigated microservice identification approaches based on the used identification technique.

2. **Automation:** process automation refers to how much an approach requires human interventions [124]. An approach can be automatic, semi-automatic, or manual.

3. `https://github.com/OAI/OpenAPI-Specification`

Table 2.3 – *Classification of the investigated microservice identification approaches based on their identification technique*

Approach	Microservice identification technique				
	Relying on genetic algorithms	Using clustering algorithms		Synthesis of virtual machines/ containers images	Hybrid identification
		Graph-based clustering	Similarity based clustering		
Gysel et al. [66]		✓			
Levcovitz et al. [94]		✓			✓
Kecskemeti et al. [86]				✓	
Chen et al. [44]		✓			✓
Mazlami et al. [100]		✓			
Baresi et al. [25]			✓		
Amiri [15]	✓			✓	
Jin et al. [76]			✓		
Jin et al. [77]					✓

(a) **Automatic:** fully automatic approaches do not require any human interventions. Actually, there are no entirely automated approaches [124]. However, similarly to Shatnawi [124], we consider the ones that human experts do not profoundly impact their results as fully automatic. Such approaches do not benefit from the knowledge of experts when it is available, even though this knowledge can enhance the identification results. Examples of fully automatic approaches are presented in [100, 15, 76].

(b) **Semi-automatic:** semi-automatic approaches need human experts to produce results. Their interactions are limited and well-specified. Nevertheless, since they are necessary, they limit the applicability of these approaches. Furthermore, the interactions are not narrowed to providing relevant, simple, and well-defined recommendations to enhance the identification results. Actually, experts perform some steps of the approach. For example, in [44], they intervene in the first step of the approach to build a DFD. While, in [66], human intervention is needed to provide a machine-readable representation of the required inputs.

(c) **Manual:** manual approaches rely entirely on human experts [124]. They present steps to systematically produce the desired outputs using the required inputs manually. Applying such approaches can be tedious, error-prone, and time-consuming. Among the investigated approaches, only one is manual [94].

3. **Direction:** an identification process can follow three directions. They are top-down, bottom-up, and hybrid.

(a) **Top-down:** the processes following a top-down direction analyze high-level software artifacts to recover lower level ones [124]. For instance, in [44], microservices are identified from the business logic of a system, described using a natural language. Another top-down approach has been proposed in [66]. It identifies microservices relying on system specification artifacts (e.g., ERMs, use cases, etc.).

(b) **Bottom-up:** bottom-up processes start by analyzing low-level software artifacts, such as source code, to produce higher level ones [124], like microservices. For instance, in [76, 77], execution traces are analyzed and clustered to identify microservices.

(c) **Hybrid:** based on the available inputs and desired outputs, a combination of the two previous directions can be adopted [7]. For example, a process can start from software requirements, following a top-down direction, and from the source code, pursuing a bottom-up path, to produce the desired output [124]. None of the investigated approaches follow a hybrid direction.

4. **Analysis technique:** several analyses can be performed by microservice identification approaches. These analyses are static, dynamic, conceptual, and hybrid.

   (a) **Static analysis:** static analysis operates without executing a software system. It examines source code while considering all the possible behaviors that might occur at run-time [54]. The main advantage of static analysis is its dependence on source code only [124]. Nevertheless, some information can not be obtained statically, such as the one related to polymorphism and dynamic binding. None of the investigated identification approaches use static analysis.

   (b) **Dynamic analysis:** dynamic analysis is performed by running a software system and then examining its executions [54]. It can inspect the actual and accurate run-time behavior of the system. The executions should be carried out relying on sufficient test cases to maximize the explored functionalities of the system and increase code coverage [10]. This analysis addresses polymorphism and dynamic binding. The principal challenge when dynamically analyzing a system, is determining the cases that covers all its functionalities. In [76], microservices are identified relying on dynamic analysis.

   (c) **Conceptual analysis:** conceptual analysis operates by examining the contents and semantics of source code files through information retrieval techniques [98, 111]. In literature, several metrics

have been proposed based on the semantic information shared between source code elements (e.g., conceptual coupling [111], conceptual cohesion [98], etc.). This analysis can be applied on any textual inputs. In [100], Mazlami et al. have proposed a coupling strategy that relies mainly on the semantic similarity between the classes of an OO source code to identify microservice. Similarly, in [25], microservices are identified based on the semantic similarity between the system operations described in OpenAPI specifications.

(d) **Hybrid analysis:** any combination of the previously presented analyses is considered as a hybrid one. Usually, they can complement and support one another by providing information that would otherwise be missing. For example, static and dynamic analyses can be combined to identify dependencies between the classes of an OO source code [10]. On the one hand, static analysis recovers static dependencies, such as method calls, by examining the code. On the other hand, dynamic one identifies runtime dependencies, for instance, the ones related to polymorphism and dynamic binding. Among the investigated approaches, only one [77] relies on a hybrid analysis (i.e., dynamic and conceptual). In this approach, execution traces are firstly analyzed to identify functional atoms, as explained earlier in this section. After that, the generated atoms are grouped based on the calls between the methods of their classes, specified in the execution traces, as well as the set of textual terms presented in class identifiers (i.e., conceptual information).

5. **Validation:** to validate the investigated microservice identification approaches, several methods were adopted. These methods are large scale experimentations and extended experimentations.

   (a) **Large scale experimentations:** large scale experimentations are usually conducted to answer a well-specified set of research questions. These questions depend mainly on the proposed approach, as well as its tackled problems. Generally, large scale experimentations start with collecting enough data from the produced results by applying the approach on sufficient case studies. Once data are gathered, they are analyzed to answer the research questions. Examples of approaches validated relying on large scale experimentation are presented in [100, 25].

   (b) **Extended experimentations:** when it is not possible to collect sufficient case studies, the validation of an approach can be performed using a limited number of them. Examples of approaches vali-

dated based on extended experimentations are presented in [66, 94, 44, 15].

6. **Tool support:** the availability of a tool for any approach increases the chances of using this approach. For instance, when identifying microservices, software architects can use different tools, developed for distinct approaches. Then, they compare the produced results and choose the one that suits them. Moreover, when a tool is backed up with documentation and a well-structured user interface, allowing to interact with it and visualize results, the chances of using it are even higher. Our investigation revealed a lack of tools for microservice identification. Some approaches provide a prototype implementation, such as the ones presented in [25, 100, 66].

Table 2.4 shows the classification results of the investigated approaches based on the remaining process aspects (i.e., automation, direction, analysis technique, tool support, and validation). "A" and "NA" respectively refer to accessible online and not accessible online.

— **Outputs of microservice identification approaches:** the produced microservices by the investigated approaches can be classified into three categories:

1. **Design-time microservices:** they are identified from documentation. Examples of approaches recovering design-time microservices are presented in [44, 25, 15].
2. **Unpackaged microservices:** they are the microservices identified from source code without packing them, such as the ones identified in [100]. Note that microservice packaging refers to performing the necessary transformations on the source code of the analyzed application to concertize the identified microservices, and thus they become executable and deployable entities. An example of such transformation is attaching the code of the identified microservice to a container, such as Docker. Another example is transforming the dependencies between classes belonging to different microservices into dependencies via REST Interfaces.
3. **Packaged microservices:** these microservices are obtained by performing any of the mentioned transformations above on the identified microservices. They have been identified in [94, 86, 76, 77].

Table 2.5 presents the classification results of the investigated microservice identification approaches based on their produced outputs.

Table 2.4 – *Classification of the investigated microservice identification approaches based on the remaining identification process aspects*

Approach	Automation			Approach direction			Analysis technique				Validation		Tool support	
	Auto.	Sem-auto.	Man.	Top-down	Bottom-up	Hybrid	Static	Dynamic	Conceptual	Hybrid	Large scale exp.	Extended exp.	Prototype	Tool
Gysel et al. [66]		✓		✓								✓	A	
Levcovitz et al. [94]			✓		✓							✓		
Kecskemeti et al. [86]		✓												NA
Chen et al. [44]		✓		✓								✓		
Mazlami et al. [100]	✓				✓				✓		✓		A	
Baresi et al. [25]	✓			✓					✓		✓		A	
Amiri [15]	✓			✓								✓		
Jin et al. [76]	✓				✓			✓			✓			
Jin et al. [77]	✓				✓					✓	✓			

Table 2.5 – *Classification of the investigated microservice identification approaches based on their outputs*

Approach	Approach outputs		
	Design-time microservices	Unpackaged microservices	Packaged microservices
Gysel et al. [66]	✓		
Levcovitz et al. [94]			✓
Kecskemeti et al. [86]			✓
Chen et al. [44]	✓		
Mazlami et al. [100]		✓	
Baresi et al. [25]	✓		
Amiri [15]	✓		
Jin et al. [76]			✓
Jin et al. [77]			✓

Discussion

This section discusses the obtained findings from our taxonomy of the investigated microservice identification approaches. The findings are organized relying on whether approaches are structure-based or task-based, additionally to their objective, required inputs, applied process, and produced outputs.

— **Structure-based versus task-based identification:** most of the investigated microservice identification approaches are structure-based (77.78%). Potentially, this is because the availability of the artifacts to be partitioned into microservices by the structure-based identification is usually higher than the task-based one.

— **Objective of microservice identification approaches:** the most targeted objective based on the taxonomy is scalability (66.67%). The second place goes to maintainability and evolution (44.44%). Cloud migration is in third place (33.33%), followed by understanding and adoption of DevOps practices (22.22%).

— **Inputs of microservice identification approaches:** almost half of the investigated approaches rely mainly on documentation to identify microservices (44.44%). These approaches can be used for both the development of new systems from scratch (i.e., forward engineering) or the migration of existing ones. Roughly all of these approaches use documentation that expresses data manipulations and exchanges between the different entities of a system. All the approaches requiring human experts (44.44%) rely on them to perform some steps (33.33%) of the approach, if not all of them (11.11%). 22.22% of the investigated approaches consider source code as the main artifact for microservice identification. Nevertheless, only 11.11% of them take into account the relationships related to data manipulations. 22.22% of the approaches have as inputs the executable system and its test cases. Only

11.11% of the approaches identify microservices from monolithic services deployed in cloud VMs.

Even though almost half of the investigated approaches rely on documentation, they can not be used to migrate existing systems unless their documentation is available and up to date. Similarly, the approaches requiring human expert intervention can not always be applied. The applicability of the ones relying on the source code, which is always available for a system to be migrated, is potentially higher.

— **Process of microservice identification approaches:** the process applied by identification approaches has many aspects: microservice identification technique, automation, direction, analysis technique, validation, and tool support.

Firstly, several techniques are used to identify microservices. Each of them manipulates the required inputs to produce the desired outputs. The investigated approaches rely mainly on clustering algorithms (66.67%). More precisely, 44.44% of them use graph-based clustering, whereas 22.22% utilize similarity-based clustering. Genetic algorithm, virtual machines/containers images synthesis and hybrid identification are the least used (11.11%). It is noteworthy that when these techniques identify microservices from the source code, the used function are limited and do not consider all the "semantics" of the concept "microservice".

Secondly, fully-automated approaches are preferred than semi-automatic and manual ones. 55.56% of the investigated approaches are automatic, 33.33% are semi-automatic, while only 11.11% are manual. Thirdly, the selection of a process direction depends on the required inputs. If it is documentation, then a top-down process is applied. This is the case with 44.44% of the investigated approaches. When the input is the source code, a bottom-up process is used (44.44%). Fourthly, conceptual analysis is the most used (22.22%), followed by dynamic and hybrid ones (11.11%). None of the investigated approaches analyze statically the code. Fifthly, large scale experimentations and extended experimentations are evenly used to validate the investigated approaches (44.44%). Finally, only 33.33% of the approaches provide a prototype accessible online. Moreover, 11.11% of them have developed tools.

— **Outputs of microservice identification approaches:** since almost half of the investigated approaches rely mainly on documentation, 44.44% of the identified microservices are design-time ones. 44.44% of the approaches produce packaged microservices, whereas 11.11% identify unpackaged ones.

To Keep in Mind

The classification results of the investigated microservice identification ap-

proaches revealed the following main limitations:

— When microservices are automatically identified from source code [94, 100], existing approaches [100] rely on limited functions that do not consider all the semantics of microservices, especially data autonomy. Therefore, the produced results by these approaches may not match those that can be manually identified by an expert. The only approach [94] that identifies microservices from source code while considering data autonomy is a manual one.

— Existing approaches either do not benefit from expert knowledge [25, 100, 15, 76, 77] or require expert intervention to perform some of their steps [66, 86, 44], if not all of them [94], which is tedious, error-prone and time-consuming. Moreover, it limits their applicability. None of the investigated approaches rely on a well-specified set of recommendations that can be easily provided by an expert and allows to enhance the identification results. Moreover, none of them combine expert recommendations with source code information.

— Existing approaches are based mainly on clustering algorithms [66, 94, 44, 100, 25, 77].

2.2.2.2 Workflow Extraction Approaches

This section firstly presents workflows. After that, similarly to microservice identification approaches, since the life cycle of a workflow extraction approach is composed of four elements, the investigated extraction approaches are classified relying on four dimensions. They are the targeted objective, the required inputs, the applied process, and the desired outputs. Figure 2.5 shows our taxonomy scheme.

Workflows as a Basis for Task-based Identification

As explained earlier, a task-based identification of microservices can be carried out based on the analysis of workflows or their constituents. Initially, workflows were introduced in companies to automate administrative procedures, in which documents are passed between different participants to achieve an overall business goal [70]. After that, they have experienced an expansion to other areas (e.g., software engineering, bioinformatics, etc.). Initially, the term workflow has been defined by the WorkFlow Management Coalition (WFMC) [70] in 1995 as follows:

"A workflow is the computerized facilitation or automation of a business process, in whole or part."

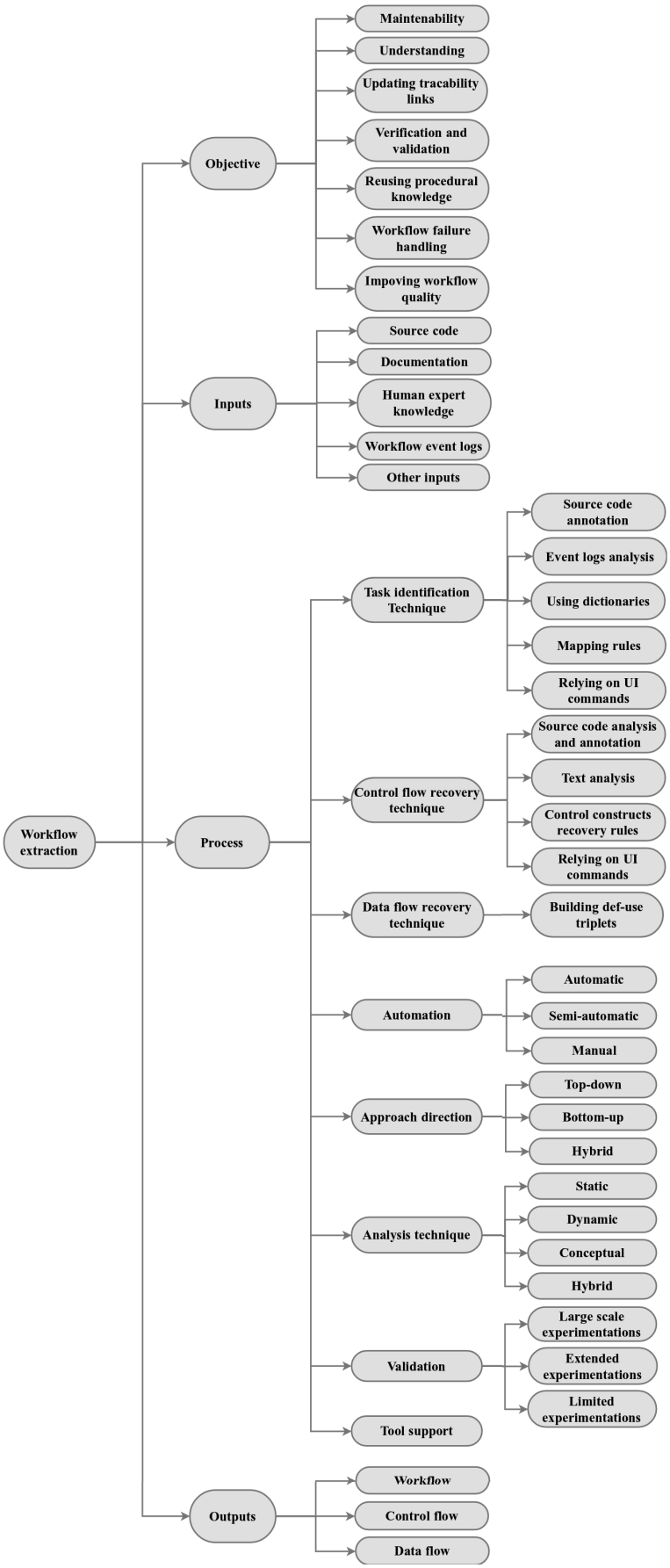

Figure 2.5 – *Taxonomy scheme of workflow extraction approaches*

This standard definition proposes a reference model for creating, deploying, and controlling workflow applications. It refers to a workflow as a solution for business processes automation. A process is defined as a coordinated sequence of a set of tasks leading to a determined result. Coordination specifies the sequencing mode of tasks (i.e., control flow), as well as the data exchanged between them (i.e., data flow). In our dissertation, these concepts are defined as follows:

— **Task:** a task is the basic composition unit of a workflow. It is the smallest execution unit, which represents an execution stage in the entire application. Its implementation does not depend on other tasks, and it is possible to reuse it in different contexts. It can define input and output data. Input data represent data needed to execute the task, whereas output data, the produced ones by this execution. A task can either be primitive or composite. A composite one may be seen as a sub-workflow consisting of primitive or composite tasks.

— **Control flow:** a control flow specifies the execution order of tasks via different constructs such as sequences, conditional branches (if and switch), and loops (for and while). In a sequence construct, a task is enabled once the execution of its predecessors is completed. In conditional branches, based on a condition (i.e., predicate) evaluation, one of several branches is chosen. In loops, tasks can repeatedly be executed.

— **Data flow:** a data flow describes data dependencies between tasks. If the outputs of a task T_i are inputs of a task T_j, then T_j cannot be executed until T_i produces its outputs. Thus, the execution of T_j depends on that of T_i.

Figure 2.6 shows an example of a workflow describing order processing. An order can be either accepted or rejected depending on the availability of the ordered items. If it is accepted, the required information is filled in, an invoice is sent, payment is approved, and the order is shipped. Otherwise, the order is closed. As illustrated in Figure 2.6, this workflow consists of six tasks. On the one hand, the control flow specifies that tasks *Fill order*, *Send invoice*, *Accept payment* and *Ship order* are executed sequentially, only if the condition *"Acceptance notice = true"* is fulfilled. On the other hand, the data flow indicates that each of the tasks *Receive order* and *Send invoice* produces one output data (resp., *Acceptance notice*, and *Invoice*). The data *Acceptance notice* is utilized to evaluate the condition *"Acceptance notice = true"*, whereas *Invoice* is used by the task *Accept payment*.

Multi-dimensional Taxonomy

— **Objective of workflow Extraction approaches:** the targeted objectives by the investigated workflow extraction approaches are the following:

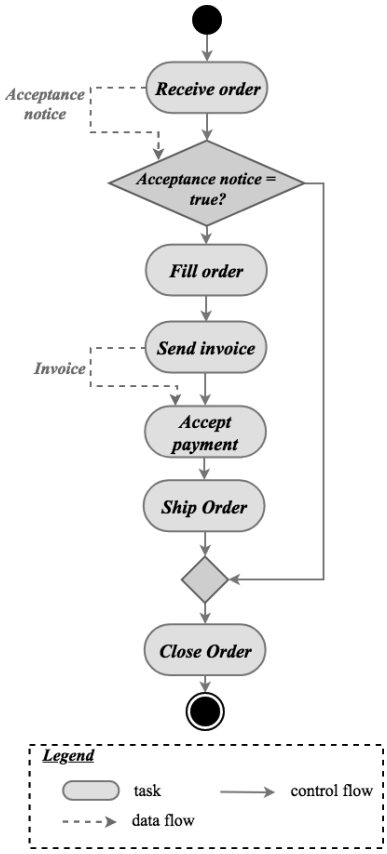

Figure 2.6 – *Example of a workflow*

1. **Maintainability:** having insight into the behavior of an application can largely enhance its maintainability. Since workflows express the behavior (i.e., the execution order of different tasks and their exchanged data), Schur et al. [119] have proposed a fully automated, though configurable tool, named Process Crawler (ProCrawl), that extracts workflows from web applications to improve their maintainability.
2. **Understanding:** commonly, software engineers rely on documentation to understand the functionalities, high-level design, and implementation details of an existing application [131]. Unfortunately, the documentation is generally outdated [131]. Some approaches aiming to improve the understanding of existing application use reverse engineering to generate updated documentation. For instance, in [90], the authors have proposed an approach to recover activity diagrams, which can represent control flows, from annotated C++ code.
3. **Updating traceability links:** tractability links between documentation and source code are important for various software engineering activities, such as system understanding and maintenance [97]. Usually,

these links are established at the beginning of the development of an application. Both documentation and source code are subject to constant change and may evolve independently. Hence, over time, the traceability links between them can drift away from the ones established initially. Nevertheless, they can be updated. For example, in [148], Zou et al. have proposed an approach to update the traceability links between the as-specified workflow (i.e., documentation) and the as-implemented one (i.e., source code).

4. **Verification and validation:** verification and validation (V & V) enable determining whether software requirements are implemented correctly and completely or not [139]. Verification evaluates the software during each life cycle phase to guarantee that the requirements set in the previous one are met [138]. Whereas, validation involves testing an application or its specification at the end of the development to guarantee that the requirements are fulfilled [138]. One of the challenges is determining the test cases that enable a thorough validation of an application. Several approaches [33, 99, 43] have been proposed to generate them based on data flow analysis. Besides this, having insight into the behavior of the application can support its validation. For this reason, workflows have been extracted from web applications in [119].
5. **Reusing procedural knowledge:** procedural knowledge, also known as how-to knowledge, specifies how to reach a particular goal or perform a specific operation via a sequence of steps [118]. An example of such knowledge is an assembling procedure of a desk outlined in a manual. Through the web, people describe their own experiences in solving particular problems (e.g., do-it-yourself instructions for house repair, etc.) or achieving specific goals (e.g., make a business card, etc.) [118]. They share it with others to reuse it in forums, blogs, Question-Answer websites, and so on [109]. This knowledge is usually represented textually. Modeling it, for instance using workflows, can promote and facilitate its reuse.
6. **Workflow failure handling:** while executing a workflow, failure may take place at any time. One of the possible ways to handle it and ensure a reliable execution is defining a transactional flow [61, 62]. It focuses on failure handling from a business point of view by specifying the changes occurring in the control flow due to task failure [61]. For instance, if a task T_1 failed, the task T_2 will be canceled. Some approaches [61, 62] recover the transactional flow from event logs.
7. **Improving the quality of a workflow:** workflows are usually studied and designed by workflow and modeling languages experts [72]. According to Ihaddadene [72], the quality of workflows can be enhanced using their event logs by better detection of parallelism, loops, and du-

Table 2.6 – *Classification of the investigated workflow extraction approaches based on their objective*

Approach	Maintainability	Under-standing	Updating traceability links	Verification and validation	Reusing procedural knowledge	Workflow failure handling	Improving workflow quality
Chen and Kao [43]				✓			
Buy et al. [33]				✓			
Martena et al. [99]				✓			
Zou et al. [148]			✓				
Gaaloul and Godart [61]						✓	
Zou and Hung [147]			✓				
Kor-shunova et al. [89]				✓			
Ihaddadene [72]							✓
Gaaloul et al. [62]						✓	
Schu-macher et al. [118]					✓		
Schur et al. [119]	✓	✓		✓			
Kosower and Lopez-Villarejo [90]		✓	✓				

plicated tasks.

Table 2.6 presents the classification results of the investigated workflow extraction approaches based on their objective.

— **Inputs of workflow extraction approaches:** the required inputs by extraction approaches can be source code, documentation, human expert knowledge, and workflow event logs.

1. **Source code:** due to its availability, the source code is the most commonly used artifact in the investigated approaches [43, 33, 99, 148, 147, 89, 90]. As explained earlier, by workflow extraction, we refer to the recovery of the entire workflow or its constituents. For instance, the proposed approach in [148] and its enhancement [147] extract workflows from the source code. Whereas, in [89, 90], activity diagrams, that can represent control flows, are recovered from C++ code.
2. **Documentation:** as explained earlier, any text, model, or illustration associated with a software system aiming to explain how it operates or

how can it be used is considered as documentation [110]. For example, in [119], workflows were extracted from Document Object Model (DOM) trees of a web application. These trees represent user interfaces (UIs).

3. **Human expert knowledge:** only one approach [90] relies on human experts. In this approach, programmers intervene by annotating the source code, which is then analyzed to recover activity diagrams.
4. **Workflow event logs:** workflow event logs contain information about the workflow as it is actually executed [135]. It is assumed possible to record events such that 1) each one refers to a task or an execution of the workflow, and 2) events are completely ordered [135]. Thus, they can be used to extract workflows. Examples of approaches requiring event logs as inputs are presented in [61, 62, 72].
5. **Other inputs:** some approaches require other inputs. For instance, in [72], the minimum and maximum durations needed to execute each task are considered available. Another example, in [119], recovering a workflow requires specifying system actors (e.g., vendor, customer, etc.), as well as a start action executed by one of them (e.g., purchase an article, etc.). The input can also be a semi-structured text, such as the one required in [118]. It represents a textual description of instructions (i.e., how to do certain things following a sequence of steps). In this case, since the text is not associated with a software system, it is not considered as documentation.
6. **Combination of multiple inputs:** some investigated approaches use a combination of the mentioned inputs [72, 119, 90]. This combination is mandatory. For instance, if a human expert does not annotate the code, it is not possible to extract activity diagrams using the proposed approach in [90].

Table 2.7 presents the classification results of the investigated workflow extraction approaches based on the software artifacts required as inputs.

— **Process of workflow extraction approaches:** the process of an extraction approach refers to 1) the way this approach achieves its objective using the required inputs and 2) how it is validated. The process is composed of eight aspects: 1) task identification technique, 2) control flow recovery technique 3) data flow recovery technique, 5) automation 6) direction, 7) analysis technique, 8) validation and 9) tool support.

1. **Task identification technique:** the adopted technique by the investigated approaches to identify tasks from the required inputs can be

Table 2.7 – *Classification of the investigated workflow extraction approaches based on their required inputs*

Approach	Source code	Doc.	Human expert	Event logs	Other inputs
Chen and Kao [43]	✓				
Buy et al. [33]	✓				
Martena et al. [99]	✓				
Zou et al. [148]	✓				
Gaaloul and Godart [61]				✓	
Zou and Hung [147]	✓				
Korshunova et al. [89]	✓				
Ihaddadene [72]				✓	Min and max execution durations of tasks
Gaaloul et al. [62]				✓	
Schumacher et al. [118]					- Semi-structured text - Dictionary of verbs - Dictionary of inputs
Schur et al. [119]		Document Object Model [1] trees			- Actors - Start action
Kosower and Lopez-Villarejo [90]	✓		✓		

source code annotation, event logs analysis, using dictionaries, utilizing mapping rules, or relying on UI commands. Note that the identification of tasks include specifying their input and output data.

(a) **Source code annotation:** in this technique, each statement or sequence of statements delimited by annotations represents a task. They are usually specified by programmers. In the investigated approaches, only one [90] relies on annotations. This is potentially because annotating the code is time-consuming, especially when dealing with large applications. Moreover, it requires in-depth knowledge of the source code. This approach does not specify task inputs and outputs.

(b) **Event logs analysis:** commonly, task identifiers are specified in event logs. Therefore, tasks can be straightforwardly recovered by analyzing these logs. Among the investigated approaches, the ones that identify tasks from logs [61, 72, 62] do not determine their inputs and outputs.

(c) **Using dictionaries:** tasks can be identified from a semi-structured

text that represents procedural knowledge [118]. As explained earlier, procedural knowledge enables achieving a particular goal or performing a specific operation via a sequence of steps. Usually, verbs are used to describe these steps (e.g., add, delete, etc.). A task represents a step in the workflow. Therefore, verbs are identified as tasks. Nevertheless, not all the verbs correspond to tasks, only the ones included in the dictionary of verbs, provided as an input of the approach. Moreover, to identify the input data of each task, another dictionary specifying all the possible inputs is required. Among the investigated approaches, there are only two, presented in [118], that use such dictionaries.

(d) **Utilizing mapping rules:** when extracting a workflow from source code, a mapping rule can be used to specify the corresponding of a task in the code. An approach relying on such rule is presented in [148]. This approach can be applied only on web applications built based on controller-centric architecture [148], which uses Model-View-Controller (MVC) [63] design pattern (i.e., specific mapping rules). This constraint limit the applicability of the approach.

In this approach, each code fragment implementing business functionalities and complying with business rules is considered as a task. Commonly, business rules are associated with database access and data validation [148]. Thus, the mapping rule is the following: invoked access beans methods, task commands objects, and any other source code fragments fulfilling the criterion mentioned above corresponds to a task.

Here, access beans represent the classes manipulating (i.e., read and/or write) data stored in the database. They contain methods performing business rules, such as item validation, and trivial methods, like getters. Trivial methods are not considered as tasks. Task commands are the classes implementing business logic pieces. For instance, a task command may be responsible for ensuring that an item is available. The authors did not specify when to consider a task command object as a task (e.g., when it is created, when one of its methods is invoked, etc.).

The business logic can also be implemented as code fragments. For this reason, the mapping rule considers these fragments as possible tasks. They are recovered from the source code by eliminating the ones that do not implement business logic. For instance, the code fragments based on Java type classes (e.g., String, etc.) do not represent tasks.

In this approach, task inputs and outputs were not identified. In fact, there is a mapping rule that associates task inputs and outputs

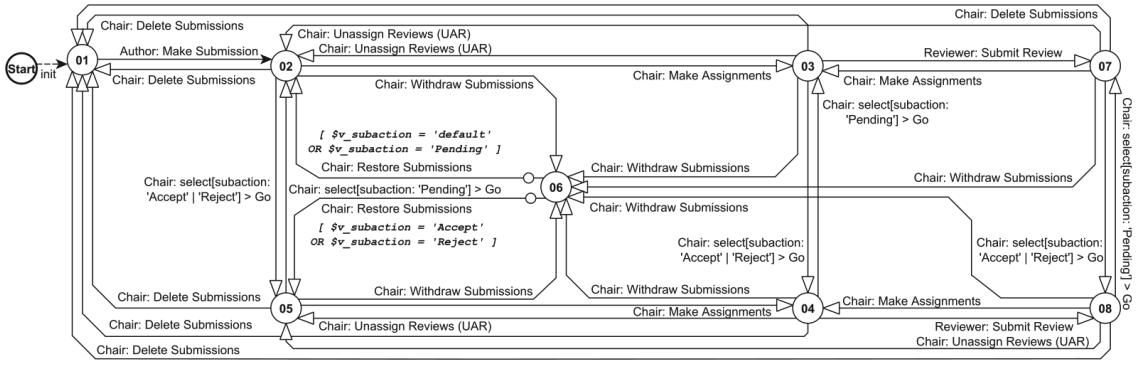

Figure 2.7 – *The extracted workflow by ProCrawl representing the peer-review process in OpenConf [119]*

with the objects of the classes *Vector*, *Enumeration*, *List*, and access beans. However, the authors did not specify how to apply this rule to identify the inputs and outputs of each task from the source code.

(e) **Relying on UI commands:** in [119], a tool that recovers a workflow from a set of UI views (i.e., technically DOM trees) of a web application has been introduced. This tool represents the recovered workflow using Extended Finite State Machines (EFSM). In an EFSM, nodes represent the abstract states of the application. They are numbered based on their detection order. State transitions denote sequences of UI commands carried out by users. Each sequence is called action.

For instance, in the example of Figure 2.7, the transition between the states 1 and 2 denotes the action *Author: Make Submission*. These actions can be viewed as tasks. It is noteworthy that the actions and their execution order (i.e., control flow) are recovered simultaneously. More precisely, ProCrawl explores a web application iteratively and incrementally. Each iteration consists of three steps. Firstly, an action among the ones available in the current state of the system is performed. The first action is an input of the approach. Secondly, the state resulting from the performed action is determined. If a new state is detected, the set of actions that should be explored are generated by applying an abstraction function on the DOM trees. Finally, each action that changes the state of the application become a state transition between the current state and the new one.

2. **Control flow recovery technique:** to extract a control flow, the investigated approaches use one of the following techniques: source code analysis and annotation, text analysis, using control constructs recov-

ery rules, and relying on UI commands.

(a) **Source code analysis and annotation:** usually, control constructs (e.g., conditional branches, loops, etc.) are represented in the source code by control statements (e.g., if, switch, while, etc.). Therefore, these constructs can be recovered by source code analysis. For instance, in [90], the authors recovered sequence, conditional branches, loops, and parallelism constructs from annotated C++ code. The recovery relied on control statements, and annotations used to specify parallel tasks and describe conditions in a human-readable way. For example, the condition "!available" will be represented by "Item not available".

(b) **Text analysis:** in [118], the control flow is extracted from a semi-structured text that represents procedural knowledge. Nevertheless, only the sequence construct is recovered. Commonly, the textual description specifies the sequential order of tasks. Thus, the recovery of this order can be performed by conceptually analyzing the text.

(c) **Using control constructs recovery rules:** unlike task identification from event logs, the control flow is not recovered straightforwardly [61, 72, 62]. Its extraction is usually performed based on well-defined rules. Each of them specifies how to recover a given control construct. The definition of these rules depends mainly on the structure of event logs and on whether additional information can be provided or not (e.g., minimum and maximum execution duration of tasks, etc.).

(d) **Relying on UI commands:** as explained earlier in this section, only one approach [119] relies on UI commands. It identifies tasks and recovers control flow simultaneously to built an EFSM. In the constructed EFSM, tasks executed sequentially can be determined based on the states of the application, since they are created relying on their detection order. In the example of Figure 2.7, based on the states 1, 2 and 3, it is possible to determine that tasks *Author: Make Submission*, *Chair: Make Assignment* and *Reviewer: Submit Review* are executed sequentially. Conditional branches are expressed relying on the states transaction guards. Cycles in the EFSM represent loops.

3. **Data flow recovery technique:** among the investigated approaches, the ones recovering a workflow [148, 61, 147, 62, 72] specify only its control flow. In other words, none of them extract both the control flow and the data flow. However, there are some approaches [43, 33, 99] that enable building a data flow from OO source code. These approaches

[43, 33, 99] compute def-use triplets, which specify data dependencies between statements. Each triplet (var, def, use) precises the name of a variable var, the line number at which var was defined (i.e., written), and the one at which this definition is used (i.e., read). These triplets combined with the corresponding of a task in the source code (e.g., a statement or sequence of statements, a method, etc.) enable building a data flow [4].

For instance, in [90], each statement or sequence of statements is considered as a task. Supposing that the code line 10 contains the statement S_1 identified as T_1 (Figure 2.8), whereas the line 11 contains S_2, which corresponds to T_2. The computing of the triplet $(var, 10, 11)$ means that there is a data dependency between S_1 and S_2, and thus between T_1 and T_2.

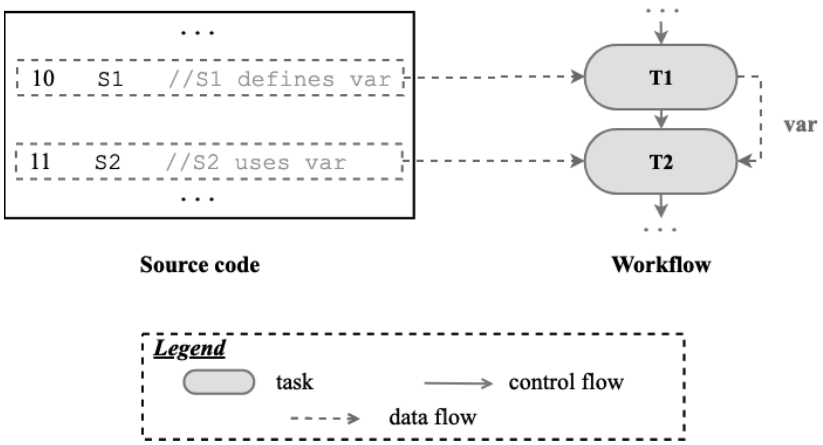

Figure 2.8 – *Example of data flow construction relying on def-use triplets*

For a primitive variable (i.e., its type is not a class), usually, definitions and uses can be determined straightforwardly [99]. An assignment to the variable represents a definition. All other operations are uses (e.g., predicate, computation, etc.). When handling objects, specifying definitions and uses cannot be performed that simply. Commonly, any operation that changes (resp., reads) the value of at least one attribute of an object is considered as a definition (resp., use). For example, in [43], an object is considered as defined in three cases: 1) it constructor is called, 2) one of its attributes is defined (i.e., its value is assigned explicitly) and 3) a method that modifies at least an attribute of the object is invoked. It is considered as used in three cases: 1) one of its attributes is used in a computation or a predicate, 2) the object is

4. More details about the construction of a data flow from def-use triplets will be presented in Chapter 4.

passed as a parameter and 3) a method that uses at least an attribute of the object is called.

Therefore, a method call can define and/or use objects. The investigated approaches computing def-use triplets rely on one of the following techniques to determine whether a call is a definition and/or a use:

— **Building DEF and USE sets:** since objects definitions and uses are determined based on whether their attributes are changed or read, the idea is to build a *DEF* and *USE* sets for each method of each class. The *DEF* (resp., *USE*) set contains the changed (resp., read) attributes by the method. A receiving object is considered as defined (resp., used) by the invoked method if its *DEF* (resp., *USE*) set includes any attributes of the receiving object. This technique was used in [43].

— **Method classification:** the methods of each class are classified based on whether they define and/or use attributes. For instance, in [99], a method can be a modifier, inspector, or inspector-modifier. This technique firstly classifies the methods of the classes having primitive attributes (e.g., int, float, etc.). Then, it moves to the ones including objects as attributes.

Table 2.8 presents the classification of the investigated workflow extraction approaches based on their task identification, control flow, and data flow recovery techniques. "SCA", "ELA", "UD","MR", "UI", "SCAA", "TA", "CCRR", respectively refer to source code annotation, event logs analysis, using dictionaries, utilizing mapping rules, relying on UI commands, source code analysis and annotation, text analysis, and using control constructs recovery rules, whereas "?" means that the needed information was not specified in the corresponding paper.

4. **Automation:** an approach can be automatic, semi-automatic, or manual. Most of the investigated workflow extraction approaches are fully automatic [43, 33, 99, 89, 148, 61, 147, 72, 62]. Only one [90] is semi-automatic. In this approach, the role of experts is limited to annotating the code. Still this limits its applicability.
5. **Direction:** an extraction process can follow three directions: top-down, bottom-up, and hybrid.
   (a) **Top-down:** there are only two approaches following a top-down direction [118, 119]. They identify workflows relying mainly on semi-structural texts [118] and DOM trees [119].
   (b) **Bottom-up:** most of the investigated workflow extraction approaches follow a bottom-up direction. They recover workflows, activity diagrams and def-use triplets from source code [43, 33, 99, 89, 90, 148, 147] and event logs [61, 62, 72].

Table 2.8 – *Classification of the investigated workflow extraction approaches based on their task identification, control flow, and data flow recovery techniques*

Approach	Task identification technique							Control flow recovery technique								Data flow recovery technique			
																Def-use triplets computation			
						Inputs/ outputs identification						Recovered control constructs				Building Def/USe sets	Method classifi-cation	Def-Use triplets	
	SCA	ELA	UD	MR	UI	In.	Out.	SCAA	TA	CCRR	UI	Sequence.	Cond. branch.	Loop	Parallel.			Prim. var.	Obj. var.
Chen and Kao [43]																✓			✓
Buy et al. [33]																	✓	✓	
Martena et al. [99]																	✓		✓
Zou et al. [148]				✓				✓				✓	✓	✓					
Gaaloul and Godart [61]		✓								✓		✓	✓	✓	✓				
Zou and Hung [147]				✓				✓				✓	✓	✓					
Kor-shunova et al. [89]	?	?	?	?	?	?	?	✓				✓	✓	✓					
Ihad-dadene [72]		✓								✓		✓	✓	✓	✓				
Gaaloul et al. [62]		✓								✓		✓	✓	✓	✓				
Schu-macher et al.[118]			✓			✓			✓			✓							
Schur et al.[119]					✓	✓					✓	✓	✓	✓					
Kosower and Lopez-Villarejo [90]	✓							✓				✓	✓	✓	✓				

(c) **Hybrid:** none of the investigated workflow extraction approaches follow a hybrid direction.

6. **Analysis technique:** several analyses can be performed by workflow extraction approaches to produce the desired outputs. These analyses are static, dynamic, conceptual, and hybrid.

   (a) **Static analysis:** most workflow extraction approaches relies on static analysis. It is used in [148, 147] to recover workflows from e-commerce applications. The authors motivate their choice by the fact that static analysis, unlike dynamic one, does not require highly skilled experts and computation resources to execute such large applications. Annotated C++ code is also statically analyzed in [90] to extract activity diagrams. In [33, 99, 43], def-use triplets are computed relying on static analysis.

   (b) **Dynamic analysis:** since event logs expose the actual run-time behavior of a system, we consider that the approaches analyzing them use dynamic analysis. These approaches are presented in [61, 72, 62].

   (c) **Conceptual analysis:** in [118], a semi-structured text was conceptually analyzed to extract a workflow.

   (d) **Hybrid analysis:** none of the investigated approaches perform a hybrid analysis.

7. **Validation:** an approach can be validated using large scale experimentations, extended experimentations, or limited experimentations.

   (a) **Large scale experimentations:** only one workflow extraction approach was validated relying on large scale experimentations [62].

   (b) **Extended experimentations:** the most commonly used validation method is extended experimentations [43, 33, 99, 148, 147, 119].

   (c) **Limited experimentations:** limited experimentations verify the validity of an approach by demonstrating that its theoretical concepts have practical potentials. Usually, it is small and may not provide a thorough validation. Examples of workflow extraction approaches validated using limited experimentations are presented in [118, 90].

8. **Tool support:** the availability of a tool for any approach increases the chances of using it. Our investigation revealed a lack of accessible tools online for workflow extraction. Several tools were developed [148, 89, 72, 62, 119], but only one, named *Flowgen* [90], is accessible. It recovers activity diagrams from annotated C++ source code. Besides tools, some approaches provide prototype implementations, such as the ones presented in [33, 99, 147].

Table 2.9 shows the classification results of the investigated workflow extraction approaches based on the remaining process aspects (i.e., automation, direction, analysis technique, tool support, and validation). "A" and "NA" respectively refer to accessible online and not accessible online, whereas "?" means that the needed information was not specified in the corresponding paper.

— **Outputs of workflow extraction approaches:** as explained earlier in this section, workflow extraction refers to recovering the entire workflow or its constituents. Therefore, the possible outputs of the investigated approaches are the following: workflow, control flow, or data flow. It is noteworthy that, since def-use triplets enable building data flows, we consider that the approaches computing them produce data flows. None of the investigated approaches express explicitly both data flows and control flows. Thus, their output is either a control flow or a data flow.
Table 2.10 presents the classification results of the investigated workflow extraction approaches based on their produced outputs.

Discussion

This section discusses the obtained findings from our taxonomy of the investigated workflow extraction approaches. The findings are organized based on the dimensions of the taxonomy: the targeted objective, the required inputs, the applied process, and the produced outputs.

— **Objective of workflow extraction approaches:** based on the taxonomy, the most targeted objective is verification and validation (41.67%). The second place goes to updating traceability links (25%). Understanding and workflow failure handling are in third place (16.67%), followed by maintainability (8,33%), reusing procedural knowledge (8,33%), and improving workflow quality (8,33%).
— **Inputs of workflows extraction approaches:** due to its availability, the source code is the most commonly used artifact by workflow extraction approaches (58,33%). Only 8.33 % of them combine it with other inputs, precisely with human expert knowledge. The second place goes to workflow event logs (25%) and other inputs (25%). Documentation and human expert (8,33%) are in third place.
— **Process of workflow extraction approaches:** the process applied by extraction approaches has many aspects:

1. **Task identification technique:** several techniques are used to identify tasks. The most utilized one is event logs analysis (25%). The second place goes to mapping rules (16.67%), followed by source code

Table 2.9 – *Classification of the investigated workflow extraction approaches based on the remaining process aspects*

Approach	Automation			Approach direction			Analysis technique				Validation			Tool support	
	Auto.	Sem-auto.	Man.	Top-down	Bottom-up	Hybrid	Static	Dynamic	Conceptual	Hybrid	Large scale exp.	Extended exp.	Limited exp.	Prototype	Tool
Chen and Kao [43]	✓				✓		✓					✓			
Buy et al. [33]	✓				✓		✓					✓		NA	
Martena et al. [99]	✓				✓		✓					✓		NA	
Zou et al. [148]	✓				✓		✓					✓			NA
Gaaloul and Godart [61]	✓				✓			✓							
Zou and Hung [147]	✓				✓		✓					✓		NA	
Korshunova et al. [89]	✓				✓		?	?	?	?		✓			NA
Ihaddadene [72]	✓				✓			✓					✓		NA
Gaaloul et al. [62]	✓				✓			✓			✓				NA
Schumacher et al.[118]	✓			✓					✓				✓	NA	
Schur et al. [119]	✓			✓				✓				✓			NA
Kosower and Lopez-Villarejo [90]		✓			✓		✓						✓		A

Table 2.10 – *Classification of the investigated workflow extraction approaches based on their outputs*

Approach	Approach outputs		
	Workflow	Control flow	Data flow
Chen and Kao [43]			✓
Buy et al. [33]			✓
Martena et al. [99]			✓
Zou et al. [148]	✓		
Gaaloul and Godart [61]	✓		
Zou and Hung [147]	✓		
Korshunova et al. [89]		✓	
Ihaddadene [72]	✓		
Gaaloul et al. [62]	✓		
Schumacher et al.[118]	✓		
Schur et al. [119]	✓		
Kosower and Lopez-Villarejo [90]		✓	

annotation (8.33%), using dictionaries (8.33%) and relying on UI commands (8.33%). Task inputs are identified in 16.67% of the approaches, whereas outputs were not determined at all.

2. **Control flow recovery technique:** control flows are recovered based on source code analysis and annotation (33,33%), control constructs recovery rules (25%), text analysis (8.33%) and relying on UI commands (8.33%). Only 25% of the approaches extract all the control constructs. Sequences (75%), conditional branches (66.67%), and loops (66.67%) are the most commonly recovered.
3. **Data flow recovery technique:** most of the investigated approaches do not extract data flows. However, as explained earlier, they can be built using def-use triplets. 25% of the approaches construct these triplets. More precisely, 16.66% of them produces triplets containing object variables, whereas 8.33% include only primitive ones.
4. **Automation:** fully-automated approaches are preferred than semi-automatic and manual ones. 91.67% of the investigated approaches are automatic, and 8.33% are semi-automatic. None of the investigated approaches are manual.
5. **Direction:** the selection of a process direction depends on the required inputs. If it is source code and event logs, then a top-down process is applied. This is the case with 83.33% of the investigated approaches. When the input is the documentation or semi-structured text, a bottom-up process is used (16.67%).
6. **Analysis technique:** static analysis is the most used (50%), followed by dynamic one (33.33%). Only few approaches rely on conceptual analysis (8.33%).

7. **Validation:** the most used validation method is extended experimentations (58.33%), followed by limited (25%), and large scale ones (8.33%).
8. **Tool support:** only 50% of the approaches provide tools. Most of them are not accessible online (41.66%). Moreover, 33.33% of these approaches have developed prototypes.

— **Extraction approach outputs:** workflows are recovered in 58.33% of the investigated approaches, whereas their constituents are extracted in 41,67% of them (i.e., data flow (25%) and control flow (16.67%)).

To Keep in Mind

The classification results of the investigated workflow extraction approaches revealed the following main limitations:

— Even though the investigated approaches identify tasks from the available artifacts, they do not always specify their input and output data. In fact, only two approaches specify task inputs [118, 119] and none of them recover task outputs. The specification of these inputs and outputs is necessary to build a data flow.

— None of the investigated approaches explicitly express both data flows and control flows. Their ultimate aim is always recovering one of them but not both.

— The fact that an approach [90] relies on expert intervention to perform any of its steps, impact negatively its applicability, especially on very large systems.

— The mapping used to extract workflows [148] from source code is specific to particular applications (i.e., based on controller-centric architecture) . Therefore, it may not be applicable on other types of applications.

— The computed def-use triplets from source code either concerns primitive variables [33] or objects [43, 99] but not both. Moreover, the ones related to objects focus mainly on the receiving object and not the ones passed as parameters.

2.2.3 Feedbacks and Learned Lessons Approaches

Now that the technical approaches have been investigated and classified, the related works concerned by feedbacks and lessons learned during the migration of existing systems towards microservices will be presented and discussed in this section. The main elements related to these feedbacks are the following:

— **Adequate granularity of microservices:** determining the proper granularity of microservices is crucial [65]. On the one hand, if they are coarse-grained, fully benefiting from their advantages is not possible. On the other hand, if they are extremely fine-grained, they might introduce additional costs and performance issues due to network latency and so on. According to Gouigoux and Tamzalit [65], *"the choice of granularity should be driven by the balance between the costs of Quality Assurance and the cost of deployment."*

— **Necessity of building autonomous teams:** adopting microservices without changing team structures can limit benefiting from their advantages [23, 42]. If teams are divided by layers (i.e., UI, application logic, and database), there are usually numerous handoffs (i.e., exchanges) between them. Fully benefiting from microservices requires removing handoffs and building autonomous teams [42]. Each team is responsible for developing, testing, and deploying its microservices independently from others.

— **Skilled and experienced developers:** to get the most out of microservices, skilled and experienced developers are needed [23]. Novice ones might miss-use tools, libraries, and so on.

— **No silver bullet:** microservices are not a silver bullet [106, 23, 42]. Their adoption will introduce new complexities and challenges [42]. Properly handling them is crucial to unlocking their benefits. Nevertheless, it can require significant costs [42]. In [106, 42, 31], some of these challenges are presented:

  — **Increased number of microservices:** decomposing monoliths into a considerable number of small microservices introduce a new complexity. For instance, traditionally, the deployment of monolithic applications involves many manual activities [42]. With the increased number of microservices, the automation of these activities is required.

  — **All the associated complexities to distributed systems**: even though a lot of lessons about well management of distributed system have been learned, it is still hard [106]. Benefiting from microservice advantages requires mastering deployment, testing, monitoring, scaling, and ensuring resilience in distributed systems [106, 42]. Moreover, data management challenges should not be forgotten. According to the CAP theorem, a distributed data store can provide simultaneously at most two of the following guarantees: consistency, availability, and partition tolerance.

2.3 Conclusion

This chapter has presented the state-of-the-art related to the migration of monolithic applications towards microservices. It is noteworthy that workflow extraction approaches are considered as related works because a task-based identification of microservices firstly extract workflows or their constituents, if they are not available, and then use them to identify microservices. The chapter started by positioning our dissertation compared to domain concepts. Then, the related works have been classified. The preliminary taxonomy classifies them into two categories: technical and feedback approaches. After that, microservice identification and workflow extraction approaches are more finely classified based mainly on four dimensions: objective, inputs, applied process, and outputs. Finally, the learned lessons and feedbacks from migrating existing systems to microservices have been presented and discussed. This chapter is concluded with the following remarks:

- A considerable number of approaches have been proposed to identify microservices or to extract workflows.
- **Remarks concerning microservice identification approaches**
  - Some microservice identification approaches do not take into consideration all the semantics of microservices, especially data autonomy. Moreover, when microservices are identified from source code, only one approach considers data autonomy.
  - The combination of software experts recommendations, when available, and source code analysis to identify microservices has not been addressed. Expert knowledge is either mandatory or not used.
  - The most commonly used algorithms to identify microservices are clustering algorithms.
- **Remarks concerning workflow extraction approaches**
  - None of the investigated approaches express explicitly both data flows and control flows.
  - Some approaches rely on expert intervention which limits their applicability.

III

Migration to Microservices: An Approach Based on Measuring Object-oriented and Data-oriented Dependencies

3.1 Introduction

Our contribution in this chapter is an approach to identify microservices from monolithic OO applications. This approach uses both architect (i.e., expert) recommendations and source code information. Nevertheless, these recommendations are not necessary, they are used when they are available. The proposed approach offers multiple ways to identify microservices. The identification can be performed using a clustering algorithm [79] or a genetic one [104]. Moreover, it can be automatic or semi-automatic, based on whether it is guided by software architect recommendations or not.

To identify microservices, our approach relies on a well-defined function that measures their quality by the assessment of their characteristics. It is based on metrics calculated from the information extracted by static analysis of the monolithic application's source code. This information relates to both the structural dependencies between the entities of the source code, as well as their dependencies via shared persistent data. The proposed quality function is considered as a similarity measure when used in our clustering algorithm, and as a fitness function when utilized in our genetic one.

This chapter is organized as follows. Section 3.2 introduces the principals of the proposed approach. Section 3.3 presents the quality measurement of microservices. Sections 3.4 and 3.5 show how our approach identifies microservices using clustering as well as genetic algorithms, and how does it inject software architect recommendations, when available, in the identification process. Finlay, Section 3.6 concludes this chapter.

3.2 Approach Principals

Our approach is based on a set of principles that constitutes the prerequisite for its applicability. They are the following:

— The type of information used to identify microservices: this information is related to the source code of the monolithic application as well as the software architect recommendations.

— The foundations of the function that evaluates the quality of potential microservices (i.e., candidates) mainly concern the analysis of the intrinsic characteristics of the concept "microservice".

— The identification process is based on two phases: extraction of the OO model of the source code, and its transformation into a microservice-based one.

— The algorithmic foundations of the approach involve using two types of algorithms: a hierarchical clustering algorithm and a search-based one (i.e., genetic).

3.2.1 Inputs of the Identification Process: Source Code and Architect Recommendations

3.2.1.1 Source Code Information

The only artifact that is necessarily available for an application is its source code. It is also the sole one reflecting the real functionalities provided by this application. Other artifacts, such as documentation, may be obsolete primarily due to the erosion phenomenon. For this reason, our approach mainly relies on static analysis of the source code to extract all the information that can be exploited for microservice identification.

In fact, our approach uses two types of information extracted by statistically analyzing the source code. The first type includes information related to the structural relationships between code entities. For instance, in OO code, these relationships include inheritance and composition between classes, methods calls, shared attributes between methods, and so on. The second type of information is related to the relationships between source code entities via shared persistent data. For an OO code, this information concerns the data accessed by each class, those (i.e., data) shared by classes, the type of operations performed on this data (i.e., read and/or write), the access frequency of a given data by a class, etc. In

our proposed identification process, all these pieces of information are analyzed and together with software architect recommendations, when available, used to identify microservices with the highest quality function values.

3.2.1.2 Recommendations of Software Architect

It is clear that nothing is worth the human expert to understand software applications. However, companies aiming to migrate their applications are often confronted with the problem of turnover and the absence of these experts. Furthermore, when they are present, their cost makes the migration process extremely expensive. This cost depends on the time spent by the experts to realize/control the migration process. Hence, clear evidence emerges regarding this process: 1) the more automated this process is, the better, 2) the migration process should use the knowledge of the experts, when they are available and 3) when utilizing experts recommendations, their form should be simple to formulate and use. Regarding the last point, the required knowledge should be related, mainly, to the use of the applications (e.g., how many microservices, which class is the center of a microservice, etc.). In our approach, we decided to exploit certain recommendations (Figure 3.1). The list of these recommendations is specified to lighten the expert task as much as possible, allowing to use its knowledge to enhance the quality of the identified microservices while reducing its cost.

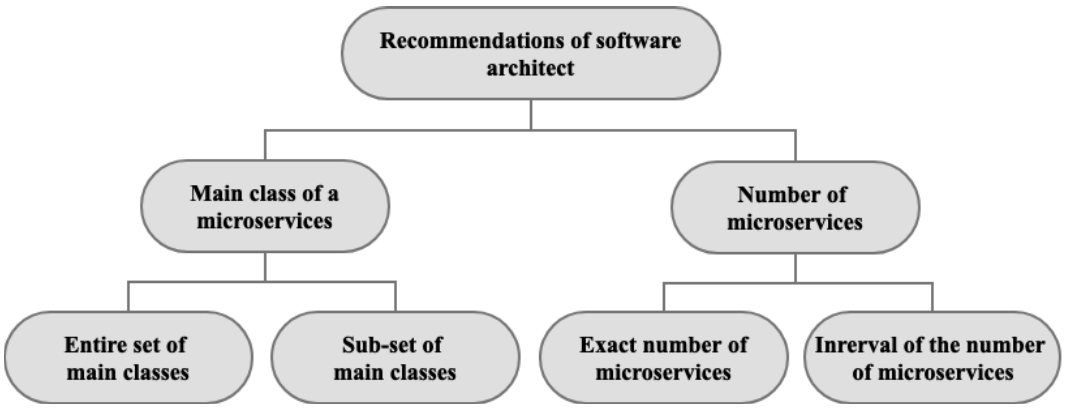

Figure 3.1 – *The used recommendations of software architect*

An important aspect that determines the quality of the identified microservices is their granularity, which strongly depends on the style of the software architect. Based on the granularity of microservices, the same application can be architected differently. Microservices can consist of small grains in one architecture and larger ones in another. In this case, the microservices of the second

architecture will simply be aggregates of the first one's microservices. The recommendations of an architect concerning the exact number of microservices of the desired architecture or the interval of their number are crucial information in relation to this question of granularity.

3.2.2 Measuring the Quality of Microservices

Our identification process partitions into clusters the entities of the original monolithic model (i.e., classes) to identify the entities of the target model (i.e., microservices). In order to determine from all the possible partitions, those that reflect a relevant microservice-based architecture, it is necessary to measure the "relevance" of this architecture. Note that the term quality of architecture or microservice is used to refer to their relevance.

The quality of a microservice-based architecture depends on the quality of its microservices. It is possible to measure it by two strategies: quantitative or qualitative. The first one measures the quality using values, whereas the second strategy allows assigning a qualification (e.g., bad, good, excellent, etc.).

In our approach, the first strategy (i.e., quantitative) was used to measure the quality of candidate microservices to select the best ones to build the target architecture. Both strategies were utilized to evaluate the obtained results by our approach during experiments.

To quantitatively measure the quality of a candidate microservice, we were inspired by the *ISO/IEC 25010:2011* [6] model, which links the quality characteristics of a software product to the corresponding metrics for measuring each one. Therefore, the characteristics that reflect the quality of a microservice were identified, and then refined to obtain the metrics that allow measuring them.

The identification of microservice characteristics is based on an analysis of their most commonly used definitions. In literature, several ones have been proposed [95, 106, 122]. Lewis and Fowler [1] [95] define microservices as "*small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies*". In [106], microservices are defined as *small*, *autonomous* services that work together. Another well admitted definition [122] considers microservices as "*self-contained (autonomous) lightweight processes communicating over HTTP, created and deployed with relatively*

1. In literature, the definition proposed by Lewis and Fowler is the most commonly used.

small effort and ceremony, providing narrowly-focused APIs to their consumers".

Based on these definitions and others [123], the main characteristics of a microservice are the following:

— **Small and focused on one functionality:** even if *small* is not a sufficient measure to describe microservices, it is used as an attempt to imply their size [95, 106, 123]. However, a question which is often asked is *how small is small?* A microservice is typically responsible for a granular unit of work (i.e., encapsulates a simple business functionality). Therefore, it is relatively small in size. It must be small enough so that its whole design and implementation can be understood. Moreover, it can be maintained or rewritten easily.

— **Autonomous:** microservices are separate entities which can be developed, tested, upgraded, replaced, deployed, and scaled independently from each other. All communications between the microservices themselves are via network calls, to enforce separation between them and ensure that they are loosely coupled (i.e., structural and behavioral autonomy) [95, 106, 122]. Moreover, each one manages its own database (i.e., data autonomy), either different instances of the same database technology, or entirely different database systems [95, 106].

— **Technology-neutral:** with a system composed of a set of collaborating microservices, it is possible to use different technologies inside each one. This allows picking the right tool for each job, rather than selecting a more standardized, one-size-fits-all approach that often ends up being the lowest common denominator [95, 106]. Additionally, different data storage technologies can be used for different parts of our system (i.e., microservices) [95].

— **Automatically deployed:** with a system consisting of a small number of microservices, it might be acceptable to manually provision machines to deploy them. Nevertheless, if this number increases, at some point, the use of a manual approach might not be possible. Hence, automatic deployment is required.

These characteristics can be classified into two categories: 1) those related to the structure and behavior of microservices, and 2) others related to the microservice development platform. The first category concerns characteristics that are independent of the microservices development platform. They are related to design (resp., identification) phase of the microservice development (resp., migration) process. It includes "small and focused on one functionality and "autonomous" characteristics. The second category depends on development platforms. It includes "technology-neutral" and "automatically deployed" characteristics, which are related to the packaging phase of the development/migration process.

In order to evaluate the quality of candidate microservices, from the above mentioned characteristics, only the ones that define microservice structure and behavior are selected: "small and focused on one functionality" and "autonomous". Note that the autonomy of a microservice includes its structural and behavioral autonomy as well as its data autonomy.

To measure these characteristics, a set of metrics were chosen and used to define a quality function. It is a weighted aggregation of two sub-functions. The first one evaluates the strength of all the structural relationships between source code elements, that will constitute the implementation of a potential microservice. The second sub-function measures the degree of dependence of microservice classes on persistent data. The goal of our approach is to identify microservices with the maximized quality function values.

3.2.3 Identification Process

A migration process includes three main phases [137]: 1) the extraction of the source code model, 2) the transformation of this original model into a new target one, and finally 3) the realization of the new target model in a target programming language.

For our problem of migrating an OO application from a monolithic architectural style to a microservice-based one, the first phase consists of extracting a model of the OO source code. It involves identifying code elements (e.g., classes, methods, etc.) and their relationships (e.g., method calls, data access, etc.). These pieces of information are extracted by a static analysis of the OO source code. The second phase aims to transform this monolithic code model into a microservice-based one. The target model entities (i.e., microservices) are identified as clusters of OO classes of the source model. To be able to identify these clusters that represent microservices in the target architecture, the quality function presented above was defined. The third step aims to generate an operational microservice-based application by packaging and deploying the identified microservices in the previous step, typically using containers.

Our approach focuses on microservice identification. Therefore, only the first and second steps of the migration process are addressed. Figure 3.2 shows the process of identifying microservices from OO source code.

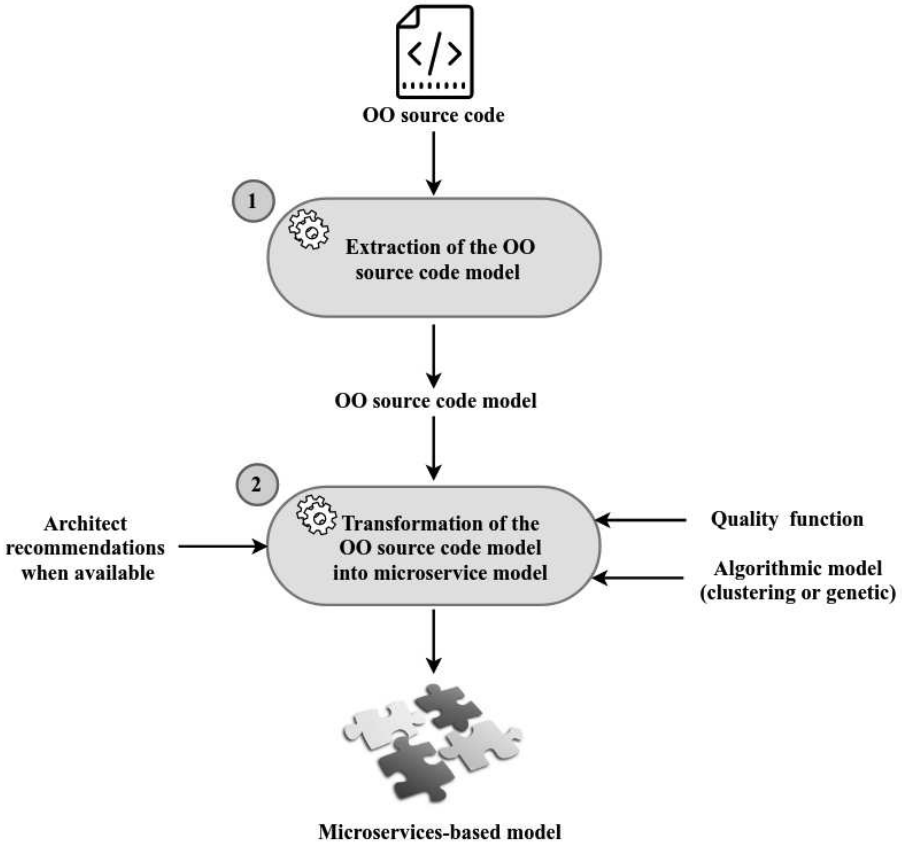

Figure 3.2 – *Process of identifying microservices from OO source code*

3.2.4 Algorithmic Foundations

As explained earlier in this section, the goal of our approach is to identify microservices with the maximized quality function values. More precisely, our aim is to group the classes of the OO application based on their quality measured using the proposed quality function. For that purpose, two types of algorithms were used: clustering and genetic algorithms.

3.2.4.1 Clustering Algorithms

Clustering algorithms partition a set of entities into clusters (i.e., groups, subsets, or categories) so that the entities of a cluster are more similar to each other than to entities belonging to other clusters [73, 142]. The measure of similarity and dissimilarity (i.e., distance) depends mainly on the features of the entities to be partitioned, which can be quantitative or qualitative, continuous or binary, nominal or ordinal [142]. For that reason, several measures have been used by different

clustering algorithms. For example, the Euclidean distance is used in K-means algorithm [68] to partition *M* points in *N* dimensions into *K* cluster, whereas cosine similarity is the most commonly utilized measure to cluster documents [142].

Among existing clustering algorithms, we chose a hierarchical clustering to identify clusters of classes that could be a "good" microservice. Our choice is motivated by the fact that, unlike some clustering algorithms that rely on the distance between entities such as K-means, hierarchal clustering is based on similarity measure [37]. Therefore, it allows us to use the proposed quality function as a similarity measure between clusters of OO classes.

3.2.4.2 Genetic Algorithms

Genetic algorithms (GAs) are meta-heuristics developed by Holland and his colleagues in the 1960s and 1970s [69]. They are based on Darwin's theory of evolution [48]. According to this theory, weak and unadapted individuals within their environment face extinction by natural selection. Whereas, the strong and well-adapted ones have higher chances to survive and pass their genes to future generations via reproduction. In the long run, better individuals (i.e., carrying the correct combination in their genes) dominate their population.

GAs apply this theory on optimization problems. To explore a solution space, each solution is encoded as an individual or a chromosome. Chromosomes are made of genes. A gene controls one or more characteristics of the chromosome [88]. GAs start with an initial population of chromosomes, which evolves at each generation by applying a set of genetic operators. These operators are mainly of three types: selection, mutation, and crossover.

In GAs, the adaptation to the environment is evaluated by a fitness function that measures the quality of each individual. This quality depends on the objectives of the optimization problem.

The solution encoding, operators, and fitness function define a specific genetic algorithm. However, the exploration of the solution space is shared between all GAs. The first step consists in determining and evaluating an initial population using the fitness function. Normally, the population is randomly initialized. Nevertheless, the better the initial population is, the better the solutions will be [37]. The second step is the generation of new solutions from existing ones. Generally, two chromosomes, called parents, are selected and crossed to generate so-called offspring children. In the third step, these children are subjected to the mutation that can modify their genes. The role of mutation is critical in GAs. In fact, crossover leads the population to converge by making the chromosomes in the

population alike. Mutation reintroduces genetic diversity back into the population and guides the search to escape from local optima [88]. The fourth step consists in evaluating the offspring children, while the fifth one selects the chromosomes of the new generation. GAs need a stopping criterion to terminate the exploration (e.g., the number of iterations, etc.).

Our goal is to define a genetic model for the problem of microservice identification and to translate this model into an algorithm enabling to select the best microservices relying on our quality function, considered as the fitness function used in GAs.

3.3 Measuring the Quality of Microservices

As explained above, the microservices constituting the target architecture are identified among a set of candidate ones. The selection criterion concerns their quality value measured by a well-defined function, which is a weighted aggregation of two sub-ones. The first sub-function measures the quality based on the structural relationships between source code entities. The second sub-function evaluates the quality of microservices relying on the data shared between these entities.

3.3.1 Measuring the Quality of a Microservice Based on Structural and Behavioral Dependencies

Regarding their structural and behavioral dependencies, the quality of microservices is measured based on two elements. The first one consists in determining the conceptual characteristics that a microservice should have and how they can be evaluated based on metrics. The second element consists in identifying among all the existing implementations of the used metrics, those that best reflect the characteristics to be assessed.

3.3.1.1 Quality Function based on the Assessment of Appropriate Characteristics

Measuring "Focused on One Functionality" Characteristic of a Microservice

In our approach, a microservice is viewed as a set of classes collaborating to provide a given function. This collaboration can be determined from source

code through the internal coupling measure, that represents the degree of direct and indirect dependencies between the set of classes, representing a candidate microservice. The more two classes of a candidate microservice use each other's methods, the more they are internally coupled. Furthermore, the collaboration can be determined by measuring the number of volatile data (i.e., not persistent data) such as attributes whose use is shared by these classes. It reflects the internal cohesion measure.

To evaluate the characteristic "Focused on One Function" of a set of classes representing a candidate microservice *M*, the function *FOne* is defined as follows:

$$FOne(M) = \frac{1}{2}(InternalCoupling(M) + InternalCohesion(M)) \tag{3.1}$$

Measuring the Structural and Behavioral Autonomy of a Microservice

As explained earlier, microservices are separate entities which can be developed, tested, upgraded, replaced, deployed, and scaled independently from each other. Therefore, in order that a set of classes represents a microservices, they should be self-sufficient. In other words, their dependencies on external classes should be minimal. This can be measured using external coupling, which evaluates the degree of direct and indirect dependencies between the classes belonging to the microservices and the external classes.

To evaluate the structural and behavioral autonomy of a candidate microservice *M*, the function *FAutonomy* was defined as follows:

$$FAutonomy(M) = ExternalCoupling(M) \tag{3.2}$$

Quality Measurement Relying on Structural and Behavioral Dependencies

The two functions that measure the quality of a microservice based on structural and behavioral dependencies were aggregated in one *FStructureBehavior* as follows:

$$FStructureBehavior(M) = \frac{1}{n}(\alpha FOne(M) - \beta FAutonomy(M)) \tag{3.3}$$

Where α and β are coefficient weights specified by a software architect and $n = \alpha + \beta$. The default value of each term is 1.

3.3.1.2 Measuring Microservice Characteristics Based on Appropriate Metrics

Earlier in this section, two microservice characteristics have been evaluated using some metrics. The rest of this section presents how these metrics are computed to reflect the semantics of the corresponding measures.

Internal coupling

The internal coupling measures the degree of direct and indirect dependencies between the classes of a microservice. These dependencies correspond to method calls. The more two classes use each other's methods, the more they are internally coupled (i.e., higher internal coupling values). This can be evaluated by measuring the frequency of internal calls between classes. Hence, the internal coupling is computed as follows:

$$InternalCoupling(M) = \frac{\sum CouplingPair(P)}{NbPossiblePairs} \tag{3.4}$$

Where *P= (C1, C2)* is a pair of classes of the microservice *M*, *NbPossiblePairs* is the number of possible pairs of classes in *M*, while *CouplingPair* is computed as follows:

$$CouplingPair(C1, C2) = \frac{NbCalls(C1, C2) + NbCalls(C2, C1)}{TotalNbCalls} \tag{3.5}$$

Where *NbCalls(C1, C2)* is the number of calls of the methods of *C1* by the methods of *C2* and *TotalNbCalls* represents the total number of method calls in the OO application.

In fact, computing internal coupling using *Equation 3.4* considers the frequency of calls between methods. Nevertheless, it does not promote clusters in which the values of the *CouplingPair* are close (i.e., all the classes are coupled). To tackle this problem, the standard deviation between the coupling values was introduced in the computing of the internal coupling as follows:

$$InternalCoupling(M) = \frac{\sum CouplingPair(P) - \sum_{PVal \in PairsVal} \sigma(PVal)}{NbPossiblePairs} \tag{3.6}$$

Where $\sigma(PVal)$ is the stand deviation between the *CouplingPair* values of the pair $PVal$ belonging to all the possible pairs of values $PairsVal$.

Note that introducing the standard deviation in the computing of the internal coupling ensures that very big or very small coupling values between a small sub-set of microservice classes do not impact the overall evaluation of the internal coupling.

External coupling

External coupling measures the degree of direct and indirect dependencies of the classes belonging to a candidate microservice on external classes. It is computed as shown in *Equation 3.7*. Where P is a pair of classes such that only one class belongs to the microservice *M*, but not both, *CouplingPair* is measured using *Equation 3.5*, $\sigma(PVal)$ is the standard deviation between the *CouplingPair* values of the pair $PVal$ belonging to all the possible pairs of values $PairsVal$. Whereas, *NbPossibleExternalPairs* is the number of pairs of classes in which only one class belong to the microservice *M*. It is noteworthy that the main difference in the evaluation of internal and external coupling is the used pairs of classes.

$$ExternalCoupling(M) = \frac{\sum CouplingPair(P) - \sum_{PVal \in PairsVal} \sigma(PVal)}{NbPossibleExternalPairs} \quad (3.7)$$

Internal cohesion

Internal cohesion evaluates the strength of interactions between classes. Generally, if the methods of two classes manipulate the same attributes, these classes are more interactive. Hence, internal cohesion is computed as follows:

$$InternalCohesion(M) = \frac{NbDirectConnections}{NbPossibleConnections} \quad (3.8)$$

Where *NbPossibleConnections* is the number of possible connections between the methods of the classes belonging to the microservice *M*, while *NbDirectConnections* is the number of connections between these methods. Two methods *method1* and *method2* are directly connected if they both access the same attribute or the call trees starting at *method1* and *method2* access the same attributes.

Note that when measuring the internal cohesion using *Equation 3.8*, the connections between the methods of the same class are taken into account. However, our goal is to evaluate the cohesion between the classes of the candidate microservices. To solve this problem, the connections between the methods of the same class are not considered. It is noteworthy that the proposed internal cohesion evaluation metric is a variation of the metric *TCC (Tight Class Cohesion)* [27].

3.3.2 Measuring the Quality of a Microservice Based on Data Autonomy

One of the main characteristics of a microservice is its data autonomy [95, 106]. A microservice can be completely data autonomous if it does not require any data from others. In order that a microservice needs less data, the internal data manipulations (i.e., reading and writing operations) between its classes should be maximized, while the accesses to external data should be minimized.

To identify such microservices, *FData* is defined as shown in *Equation 3.9*. Where α and β are coefficient weights specified by a software architect and $n = \alpha + \beta$. The default value of each term is 1. *FData* is based on measuring data dependencies between the classes of the microservice (*FIntra*), as well as their dependencies with external classes (*FInter*). Note that the microservices having high accesses to data manipulated by external classes (i.e., high values of *FInter*) are penalized. Indeed, the greater *FInter* is, the lower *FData* will be.

$$FData(M) = \frac{1}{n} \quad (\alpha FIntra(M) - \beta FInter(M)) \tag{3.9}$$

3.3.2.1 Computing FIntra

$FIntra$ function applied on a microservice M represents the average of data dependencies measurement between all the possible pairs of classes belonging to M (*Equation* 3.10). We chose the average of the data manipulations measurement instead of the sum because the latter promotes large microservices (i.e., consisting of a high number of classes) even if their classes do not manipulate the same data.

$$FIntra(M) = \frac{\sum_{c_i,c_j \in M} DataDependencies(c_i, c_j)}{NbPossiblePairsInMicroservice} \tag{3.10}$$

To better understand, Figure 3.3 shows an example, in which the microservice *M2* contains four classes (*C4*, *C5*, *C6*, and *C7*) manipulating the same data (*D2* and *D3*). Whereas, *M1* has only two classes (*C1* and *C2*) accessing to *D1*. Therefore, the sum of data manipulations measurement for *M2* will be higher than *M1*. This indicates that *M2* is better, while it is not the case, since all the classes of *M1* manipulate the same data, whereas merely a one-third of *M2*'s classes do that.

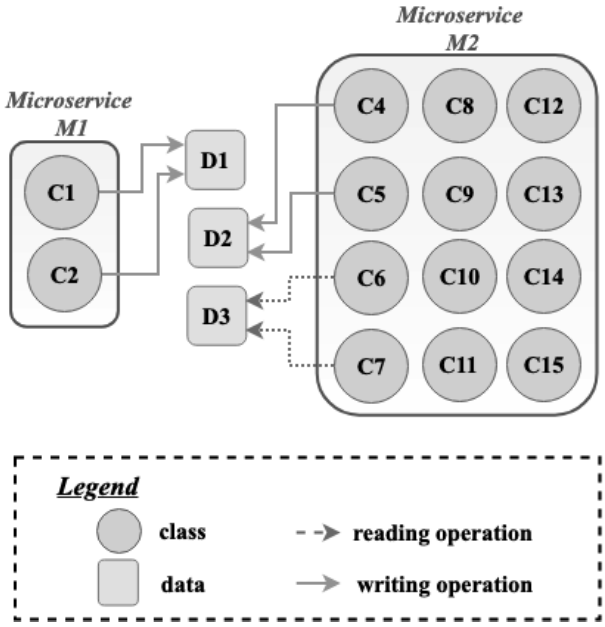

Figure 3.3 – *Example motivating the use of the average to compute FIntra*

3.3.2.2 Computing FInter

$FInter$ represents the average of measuring data dependencies between all the pairs of classes in which only one class belongs to the microservice M (*Equation* 3.11). Note that the main difference between *FIntra* and *FInter* is the used pairs of classes.

$$FInter(M) = \frac{\sum_{c_i,c_j \in Classes} DataDependencies(c_i, c_j)}{NbPossibleExternalPairs} \tag{3.11}$$

3.3.2.3 Computing DataDependencies

Both *FIntra* and *FInter* rely on *DataDependencies*. This function measures data dependencies between two classes based on their read and written data. Considering the access mode to data (i.e., read and/or write) while measuring the data autonomy of a microservice is important. Based on the data ownership pattern proposed to decompose a monolith into microservices [24], a data can be modified or created just through its owner (i.e., corresponding microservice). Other microservices are allowed to have a copy of the data that they do not own, but they should be careful about its staleness.

DataDependencies is measured as follows:

$$DataDependencies(c_i, c_j) = \frac{\sum_{k \in Data} D(c_i, c_j, k)}{NbDataManipulatedInMicroservice} \quad (3.12)$$

Where $Data$ is the set of data manipulated in the microservice *M* and *NbDataManipulatedInMicroservice* is its size. D is defined, inspired by the proposed approach in [15], as follows:

$$D(c_i, c_j, k) = \begin{cases} 1 \text{ if } c_i \text{ and } c_j \text{ write } k. \\ 0.5 \text{ if a class writes } k \text{ and the other one reads it.} \\ 0.25 \text{ if } c_i \text{ and } c_j \text{ read } k. \\ 0 \text{ otherwise.} \end{cases}$$

In fact, *DataDependencies* is the average of data manipulation measurement for a given pair of classes. We chose the average instead of the sum to promote microservices manipulating data more strongly (i.e., microservices having higher values of *D*). Note that if a class reads and writes the same data only the writing (major) operation is considered.

Figure 3.4 shows an example of two microservices *M1* and *M2*. The results of measuring *DataDependencies* for each one are the following:

— *DataDependencies(M1)=D(C1,C2,D1)/1=1/1=1*

— *DataDependencies(M2)=D(C4,C5,D1)+D(C6,C7,D2)+D(C12,C13,D3)+ D(C14,C15,D4))/4=(0.25+0.5+0.25+0.25)/4*
DataDependencies(M2)=1,25/4=0.31

In this example, while measuring *DataDependencies* only the pairs of classes for which the value of *D* is different from null were taken into account for briefness and clarity. Moreover, as can be seen, the use of the average to compute *DataDepndencies* promotes *M1*.

Measuring *DataDependencies* using *Equation* 3.12 does not take into account the frequency of data manipulations. This frequency has a substantial impact on the autonomy of the identified microservices. For instance, if a class of a candidate microservice $M1$ frequently manipulates a data $D1$ associated to another microservice $M2$ (i.e., $D1$ was associated to $M2$ because this microservice contains more classes manipulating it less often (Figure 3.5)). Once the microservice identification and packaging are done, each manipulation will be interpreted as a

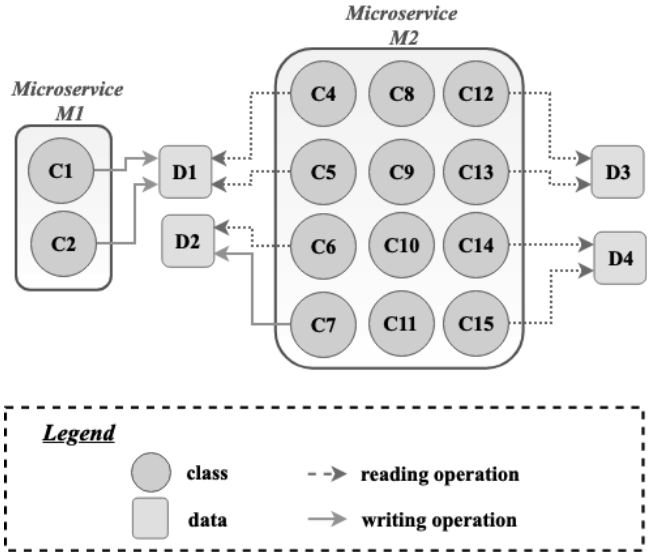

Figure 3.4 – *Example motivating the use of the average to compute DataDependencies*

communication between $M1$ and $M2$. Frequent manipulations produce frequent communications, which reduce the autonomy of the identified microservices.

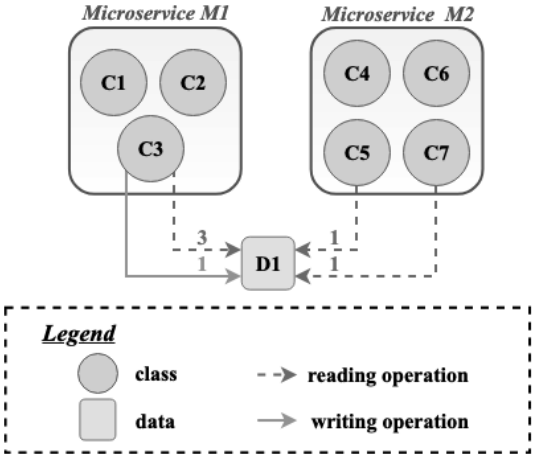

Figure 3.5 – *Example of the frequency of data manipulations*

Hence, to identify autonomous microservices, the frequency was introduced in the *DataDependencies* measurement, as shown in *Equation* 3.13.

$$DataDependencies(c_i, c_j) = \frac{\sum_{k \in Data}(D(c_i, c_j, k) * Freq(c_i, c_j, k))}{NbDataManipulatedInMicro} \tag{3.13}$$

Where $Freq(c_i, c_j, k)$ is the number of times the classes c_i and c_j manipulate *k*.

It is defined as follows:

$$Freq(c_i, c_j, k) = FreqCl(c_i, k) + FreqCl(c_j, k) \tag{3.14}$$

Measuring the frequency using *Equation* 3.14 does not promote clusters in which the data manipulation frequency of *k* for the two classes c_i and c_j is close. For instance, in the example of Figure 3.6, the classes of the microservice *M1* manipulates *D1* with almost the same frequency, unlike the classes of the microservice *M2*.

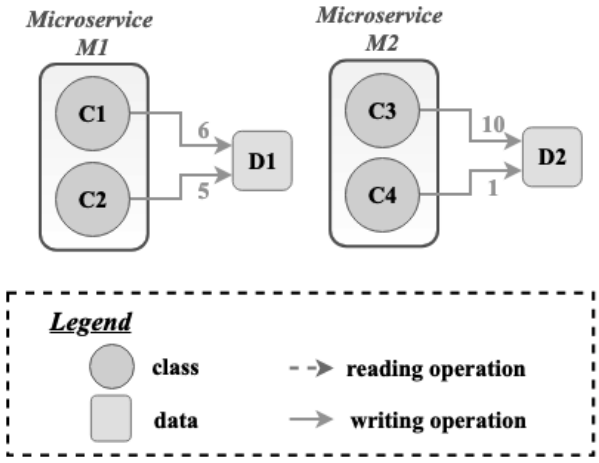

Figure 3.6 – *Example motivating the introduction of standard deviation in the computing of Freq*

To tackle this problem, the standard deviation was introduced in the frequency measurement, as shown in *Equation* 3.15.

$$Freq(c_i, c_j, k) = FreqCl(c_i, k) + FreqCl(c_j, k) - \sigma(FreqCl(c_i, k), FreqCl(c_j, k)) \tag{3.15}$$

Example of FData computation

To better understand, let us measure *FData* of the microservice *M1* consisting of the classes *C1* and *C2*, as shown in Figure 3.7.

There is only one possible pair of classes belonging to *M1*: *(C1,C2)*. Whereas, the possible pairs in which only one class is in $M1$ are *(C1,C3)*, *(C1,C4)*, *(C1,C5)*, *(C2,C3)*, *(C2,C4)* and (C2,C5).

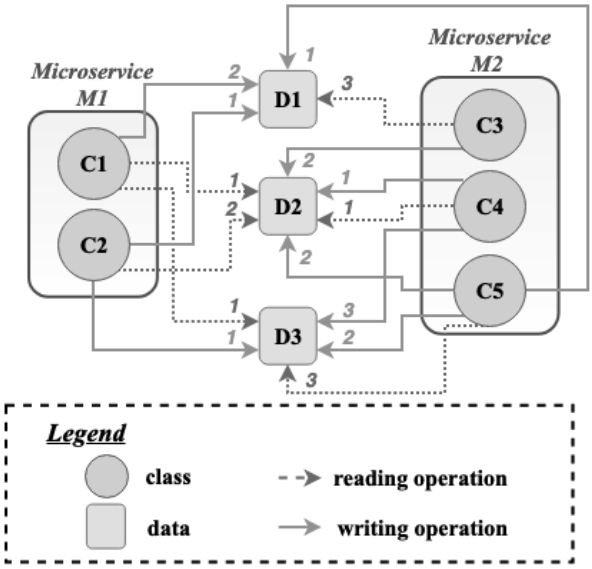

Figure 3.7 – *Example used to compute FData*

To evaluate *FData*, the values of *D* and the frequency of data manipulations were computed. The results are shown in tables 3.1 and 3.2 respectively. Based on these computations, *FData* is measured as follows:

— *DataDependencies(C1,C2)= (D(C1,C2,D1)*Freq(C1,C2,D1)+D(C1,C2,D2)* Freq(C1,C2,D2)+D(C1,C2,D3)*Freq(C1,C2,D3))/3 =(2.5+0.63+1)/3*
DataDependencies(C1,C2)=1.38

— *FIntra(M1)=DataDependencies(C1,C2)/1=* 1.38

— *FInter(M1)= (DataDependencies(C1,C3)+DataDependencies(C1,C4)+ DataDependencies(C1,C5)+DataDependencies(C2,C3)+ DataDependencies(C2,C4)+DataDependencies(C2,C5))/6 = (0.42+0.83+1.67+1.17+1.42+2.17)/6*
FInter(M1)=1.28

— *FData(M1)= (1*FIntra-1*FInter)/2= 1.38-1.28/2=0.05*

Table 3.1 – *Measurement of D between the classes of the example shown in Figure 3.7*

	D1					D2					D3				
	C1	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**
C1	/	1	0.5	0	1	/	0.25	0.5	0.5	0.5	/	0.5	0	0.5	0.5
C2	1	/	0.5	0	1	0.25	/	0.5	0.5	0.5	0,5	/	0	1	1
C3	0.5	0.5	/	0	0.5	0.5	0,5	/	1	1	0	0	/	0	0
C4	0	0	0	/	0	0.5	0.5	1	/	1	0.5	1	0	/	1
C5	1	1	0.5	0	/	0.5	0.5	1	1	/	0.5	1	0	1	/

Table 3.2 – *Measurement of Freq between the classes of the example shown in Figure 3.7*

	D1					D2					D3				
	C1	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**	**C1**	**C2**	**C3**	**C4**	**C5**
C1	/	2.5	4.5	0	2.5	/	2.5	2.5	2	2.5	/	2	0	3	2.5
C2	2.5	/	3	0	2	2.5	/	4	2.5	4	2	/	0	3	2.5
C3	4.5	3	/	0	3	2.5	4	/	2.5	4	0	0	/	0	0
C4	0	0	0	/	0	2	2.5	2.5	/	2.5	3	3	0	/	4.5
C5	2.5	2	3	0	/	2.5	4	4	2.5	/	2.5	2.5	0	4.5	/

3.3.3 Global Measurement of the Quality of a Microservice

The global evaluation of the quality of a microservice depends on the measurement of its quality based on structural and behavioral dependencies, as well its quality relying on its data autonomy. To measure this quality, the function *FMicro* was defined as follows:

$$FMicro(M) = \frac{1}{n}(\alpha FStructureBehavior(M) + \beta FData(M)) \tag{3.16}$$

Where α and β are coefficient weights specified by a software architect and $n = \alpha + \beta$. The default value of each term is 1.

Note that the coefficient weights show the importance of the relationships between code entities (*FStructureBehavior*) and their relationships with persistent data (*FData*). A software architect, according to his/her knowledge of the system to migrate, can decide to give more or less importance either to *FStructureBehavior* or *FData*.

For example, if the source code was developed respecting the rules of separation of responsibility, modularity, and so on, the architect can give a high coefficient weight to *FStructureBehavior*. Otherwise, he/she can lower it. Similarly, the architect can give a high coefficient weight to *FData* if the migration process aims to partition the database, and lower them if not. Note that it is possible to use different coefficient weights, then compare the produced results and chose the best one.

3.4 Microservice Identification Using Clustering Algorithms

In order to identify microservices, our approach groups the classes of an OO application based on their quality measured using the proposed quality function

presented above. For that purpose, a hierarchical clustering algorithm [79] with our quality function as a similarity measure was defined. It partitions source code classes into clusters corresponding each to a microservice. The goal is to have in each cluster, classes that are more similar to each other.

3.4.1 Automatic Identification of Microservices Using a Hierarchical Clustering Algorithm

The automatic identification of microservices from OO source code using a hierarchical clustering algorithm consists of two steps:

— *Step 1:* in this step, the hierarchical clustering algorithm, which produces a binary tree also called dendrogram, is applied on the classes of the OO application.

— *Step 2:* this step aims to obtain disjoint clusters from the dendrogram. Each cluster is considered as a microservice.

3.4.1.1 Building a Dendrogram from OO Source Code

Hierarchical clustering algorithm groups together the classes with the maximized value of the quality function. In the beginning, each class is considered as a cluster. Then, the quality function is measured between all pairs of clusters. The clusters with the highest quality function value are merged into a new one. Next, the algorithm measures the quality function between the newly formed cluster and all others and successively merge the pair with the highest quality function value. These steps are repeated until the number of clusters equals one, as shown in Algorithm 1. Consequently, the classes of the OO source code are expressed in a hierarchical view presented in a dendrogram (Figure 3.8).

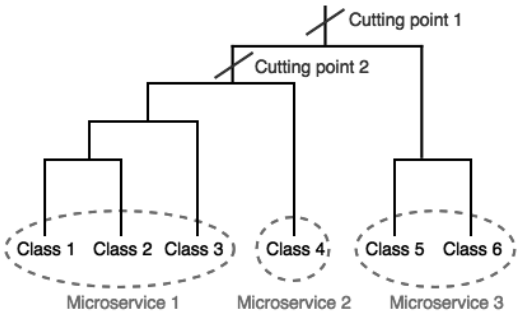

Figure 3.8 – *Dendrogram with a set of microservices*

Algorithm 1: Hierarchal clustering

input : OO source code $code$
output: A dendrogram $dendro$

let $S_{classes}$ be the set of classes extracted from the OO source code $code$;
Let $S_{Clusters}$ be a set of clusters of classes;
for *each* $class \in S_{classes}$ **do**
    let $class$ be a $cluster$;
    add $cluster$ to $S_{Clusters}$;
end
while $size(S_{Clusters}) > 1$ **do**
    Let $(cluster_1, cluster_2)$ be the closest clusters based on the quality function;
    Let $New_{cluster} \leftarrow merge(cluster_1, cluster_2)$;
    remove $cluster_1$ and $cluster_2$ from $S_{Clusters}$;
    add $New_{cluster}$ to $S_{Clusters}$;
end
$dendro \leftarrow getCluster(S_{Clusters})$;
return $dendro$;

3.4.1.2 Partitioning a Dendrogram to Obtain Microservices

So as to obtain a partition of disjoint clusters, the resulting dendrogram has to be cut at some point. To determine the best cutting point, the standard depth-first search (DFS) algorithm was used. Initially, on the root node, the quality of the current node is compared to the quality of its child nodes. If the quality value of the current node is lower than the average quality value of its children, the current node is a cutting point. Otherwise, the algorithm continues recursively through its children. Figure 3.8 shows an example of partitioning a set of classes into microservices.

3.4.2 Semi-automatic Identification of Microservices Based on a Hierarchical Clustering Algorithm

The used recommendations in our approach are mainly related to architectural aspects. Depending on the availability of these recommendations, five ways to define our clustering algorithm were proposed.

3.4.2.1 Use of Information Related to the Gravity Centers of Microservices

In many cases, a microservice is built around a class that constitutes its "gravity center". It represents the functional core of the microservice (i.e., the main

class). Other classes revolve around this "core" class to constitute the complete implementation of the microservice. This information, when available, is exploited to build microservices, where each center of gravity will be the starting point to group the classes.

Two cases were considered. The first one concerns the situation where the entire set of gravity centers is available. The second case is related to the situation where only a sub-set of these centers is known.

Microservice Identification Based on the Entire Set of Gravity Centers

To identify microservices, the idea is to consider each gravity center as a cluster. Then, the remaining classes of the OO application are partitioned iteratively on these clusters based on their quality evaluation. At each iteration, a remaining class is associated to the cluster for which adding this class produce the highest value of the quality function. In the end, the OO application classes are partitioned into clusters. Each one of them contains one gravity center (Figure 3.9). Algorithm 2 shows the presented clustering.

Algorithm 2: Clustering based on the entire set of gravity centers

input : A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
output: A set of clusters $S_{Clusters}$

```
for each class ∈ S_Gcenters do
    let class be a cluster;
    add cluster to S_Clusters;
end
for each class ∈ S_Rclasses do
    for each cluster ∈ S_Clusters do
        let tempCluster be a cluster containing all the classes of cluster;
        add class to tempCluster;
        let quality be the quality function value of tempCluster;
        save the pair (quality,cluster);
    end
    let best_cluster be the cluster of the pair (quality,cluster) such that quality is the
      highest value in all pairs;
    add class to best_cluster;
end
return S_Clusters;
```

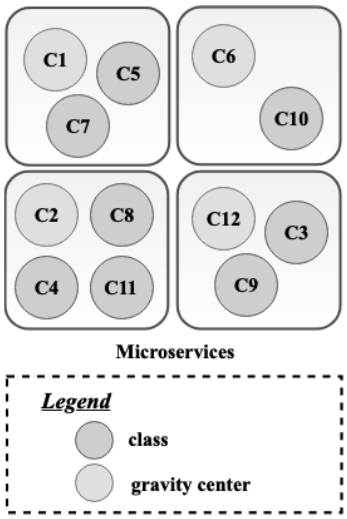

Figure 3.9 – *Identified microservices based on the entire set of gravity centers*

Microservice Identification Based on a Sub-set of Gravity Centers

In the case where only a sub-set of gravity centers is available, making use of this information is important. The question is how can this be done? To answer this question, Algorithm 3 was proposed. In this algorithm, initially, each gravity center is considered as a cluster (lines 1 to 4). Moreover, an additional one is added (lines 5 and 6). This cluster represents the pair of the remaining classes with the highest quality function value. The idea is to firstly partition the remaining classes of the OO application iteratively on these clusters based on their quality evaluation (lines 7 to 16). Therefore, all the produced clusters contain a gravity center except the additional cluster, as shown in Figure 3.10. After that, this additional cluster will be decomposed using the hierarchical clustering algorithm presented in Section 3.4.1 (lines 17 to 19).

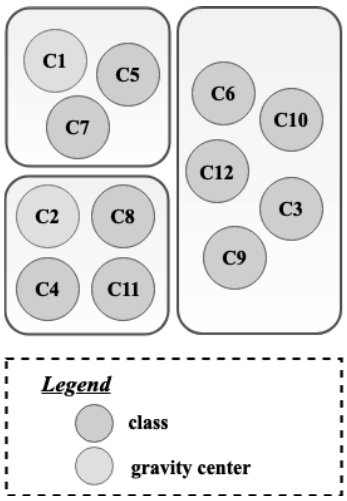

Figure 3.10 – *Preliminary partitioning of OO classes based on a sub-set of gravity centers*

Algorithm 3: Clustering based on a sub-set of gravity centers

input : A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
output: A set of clusters $S_{Clusters}$

```
for each class ∈ S_Gcenters do
    let class be a cluster;
    add cluster to S_Clusters;
end
let add_cluster be the pair of S_Rclasses with the highest quality value;
add add_cluster to S_Clusters;
for each class ∈ S_Rclasses do
    for each cluster ∈ S_Clusters do
        let tempCluster be a cluster containing all the classes of cluster;
        add class to tempCluster;
        let quality be the quality value of tempCluster;
        save the pair (quality,cluster);
    end
    let best_cluster be the cluster of the pair (quality,cluster) such that quality is the
      highest value in all pairs;
    add class to best_cluster;
end
let remaining_clusters the clusters resulting from partitioning add_cluster using the
  hierarchical clustering algorithm;
remove add_cluster from S_Clusters;
add remaining_clusters to S_Clusters;
return S_Clusters;
```

3.4.2.2 Use of Information Related to the Number of Microservices

The granularity of the microservices constituting the target architecture is a determinant element of their relevance. Nevertheless, this element is very dependent on the architect's style (i.e., coarse grains versus fine grains). For this reason, the granularity can be very variable in the approaches based only on the source code. Two pieces of information from the architect can be indicators of this granularity: the number of classes per microservice and the number of microservices in the target architecture. We believe that defining the same number of classes for all microservices goes against a partitioning that depends on their quality (i.e., in the same architecture, it is not excluded that some microservices can be relatively larger than others). Giving the exact number of classes for each microservice amounts to manually identifying them (i.e., against a semi-automatic approach). Hence, the retained recommendation as an indicator of the granularity of microservices is their number in the target architecture.

Two cases were considered. The first one concerns the situation where the exact number of microservices is known. The second case is related to the situation where only an interval of their number is available.

Microservice Identification Based on their Exact Number

To make use of the availability of the exact number of microservices, our idea is to firstly identify microservices, and then compose/decompose the identified ones to obtain the exact number, while taking into account the quality of the identified microservices. Thus, there are two steps: 1) microservices identification, and 2) microservice composition/decomposition (Algorithm 4).

Algorithm 4: Clustering based on the exact number of microservices

input : A number of microservices Nb
A set of OO classes $S_{Classes}$
A dendrogram $dendrogram$
output: A set $S_{Clusters}$ consisting of Nb cluster

```
1  let initial_clusters be the identified clusters using the hierarchical clustering algorithm;
2  if sizeOf(initial_clusters) = Nb then
3  |   S_Clusters ← initial_clusters;
4  else
5  |   if sizeOf(initial_clusters) < Nb then
6  |   |   S_Clusters ← decomposeMicro(Nb,initial_clusters,dendrogram);
7  |   else
8  |   |   S_Clusters ← composeMicro(Nb,initial_clusters,dendrogram);
9  |   end
10 end
11 return S_Clusters;
```

Step 1: Microservices identification: to identify microservices, the hierarchical clustering algorithm presented in Section 3.4.1. is used.

Step 2: Microservices composition/decomposition: once the microservices are identified, their number is compared to the exact one:

— If they are equal, the identification is completed.

— If the number of identified microservice is lower than the exact number, these microservices are decomposed. The question is which ones should be decomposed? An adequate answer to this question should take into account the quality of the produced decomposition. Therefore, our idea is to decompose a microservice at a time and compute the sum of the quality function values for the new set of microservices. The decomposition which produces the best result is chosen. This step is repeated as long as the number of microservices is lower then the requested one. Algorithm 5 shows the proposed microservices decomposition.

Algorithm 5: Decompose microservices

input : A number of requested microservices Nb_{req}
A set of clusters $S_{Clusters}$
A dendrogram $dendrogram$
output: A set $S_{Clusters}$ consisting of Nb_{req} cluster

while *sizeOf*($S_{Clusters}$)$\neq Nb_{req}$ **do**
  for ***each*** *cluster* $\in S_{Clusters}$ **do**
    let S_{temp} be a set of clusters containing all the clusters of $S_{Clusters}$;
    find $cluster$ children in $dendrogram$;
    replace $cluster$ by its children in S_{temp};
    let $sum \leftarrow \sum_{tcluster \in S_{temp}} FMicro(tcluster)$;
    save the pair (sum,$cluster$);
  end
  let $best_{cluster}$ be the cluster of the pair (sum,$cluster$) such that sum is the highest value in all pairs;
  replace $best_{cluster}$ by its children in $S_{Clusters}$;
end
return $S_{Clusters}$;

— If the number of identified microservice is higher than the requested one, instead of decomposing them, they are composed until the exact number is obtained (Algorithm 6).

Algorithm 6: Compose microservices

input : A number of requested microservices Nb_{req}
A set of clusters $S_{Clusters}$
A dendrogram $dendrogram$
output: A set $S_{Clusters}$ consisting of Nb_{req} cluster

while *sizeOf*($S_{Clusters}$)$\neq Nb_{req}$ **do**
  for ***each*** *cluster* $\in S_{Clusters}$ **do**
    let S_{temp} be a set of clusters containing all the clusters of $S_{Clusters}$;
    let $root$ be the root node of $cluster$ in $dendrogram$;
    let $cluster_2$ be the cluster having the same root of $cluster$;
    replace $cluster$ and $cluster_2$ by $root$ in S_{temp};
    let $sum \leftarrow \sum_{tcluster \in S_{temp}} FMicro(tcluster)$;
    save the pair (sum,$root$);
  end
  let $best_{cluster}$ be the cluster of the pair (sum,$cluster$) such that sum is the highest value in all pairs;
  remove $best_{cluster}$ children from $S_{Clusters}$;
  add $best_{cluster}$ to $S_{Clusters}$;
end
return $S_{Clusters}$;

Composing/decomposing microservices this way allow us, on the one hand, to obtain the exact number of microservices, and on the other hand, produce a

decomposition with the highest quality function values.

Microservice Identification Based on an Interval of their Number

In order to identify microservices using an interval of their number, the idea is to firstly choose a random number within this interval, then apply Algorithm 4 (Algorithm 7).

Algorithm 7: Clustering based on an interval of the number of microservices

input : A minimum number of microservices Nb_{min}
A maximum number of microservices Nb_{max}
A set of OO classes $S_{Classes}$
output: A set $S_{Clusters}$ consisting of Nb cluster

1 let Nb be a random number within $[Nb_{min}, Nb_{max}]$;
2 $S_{Clusters} \leftarrow Nb$ microservices identified using Algorithm 4;
3 return $S_{Clusters}$;

3.4.2.3 Microservice Identification Based on their Exact Number and a Sub-set of Gravity Centers

In the case where, in addition to the exact number of microservices, a sub-set of gravity centers is available, making use of this information is important. The question is how can it be used in combination with the number of requested microservices? The answer to this question is by combining Algorithm 3 and Algorithm 4 with some adjustments. The idea is, similarly to Algorithm 3, initially, each gravity center is considered as a cluster. Furthermore, an additional cluster is added. This cluster represents the pair of the remaining classes with the highest quality function value. In the end, the additional cluster (i.e., the cluster which does not contain any gravity center) will be partitioned using Algorithm 4 to obtain the exact number of microservices. In this case, the number of microservices produced by Algorithm 4 equals to the exact number of microservices minus the size of the sub-set of gravity centers. Algorithm 8 shows the presented clustering.

3.5 Microservice Identification Using Genetic Algorithms

This section presents our genetic model of the microservice identification problem as well as a set of algorithms implementing this model. They integrate taking into account the software architect recommendations or not.

Algorithm 8: Clustering based on the exact number microservices and a subset of gravity centers

input : A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
output: A set $S_{Clusters}$ consisting of Nb cluster

```
for each class ∈ S_Gcenters do
    let class be a cluster;
    add cluster to S_Clusters;
end
let add_cluster be the pair of S_Rclasses with the highest quality value;
add add_cluster to S_Clusters;
for each class ∈ S_Rclasses do
    for each cluster ∈ S_Clusters do
        let tempCluster be a cluster containing all the classes of cluster;
        add class to tempCluster;
        let quality be the quality value of tempCluster;
        save the pair (quality,cluster);
    end
    let best_cluster be the cluster of the pair (quality,cluster) such that quality is the
      highest value in all pairs;
    add class to best_cluster;
end
let rest_micro ← Nb - SizeOf(S_Gcenters);
let remaining_clusters the rest_micro clusters resulting from partitioning add_cluster
  using Algorithm 4;
remove add_cluster from S_Clusters;
add remaining_clusters to S_Clusters;
return S_Clusters;
```

3.5.1 A Genetic Model for the Process of Microservice Identification

To emulate the process of natural evolution, a genetic model should define several elements: the encoding of the problem, the specification of the genetic operators, the initial population and the fitness function.

3.5.1.1 Encoding of the Problem

In our approach, a GA is used to identify microservices from OO source code. Thus, a potential solution is a partition of these classes such that 1) none of its subsets is empty, 2) their union represents all the classes of the OO application

and 3) their intersection is empty. In GAs, each solution is traditionally encoded as a single chromosome consisting of one or more gene(s). In our case, a gene represents a set of classes. Figure 3.11 illustrates a chromosome with three genes.

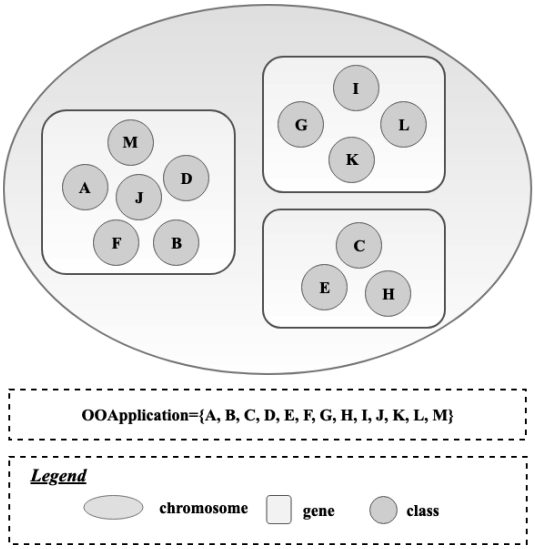

Figure 3.11 – *Encoding of a chromosome*

3.5.1.2 Definition of the Genetic Operators

Genetic operators explore a solution space by creating new solutions from existing ones. Their role is to allow the evolution of the population until an acceptable solution is reached. These operators are mainly of three types:

— **Selection operators:** there are two selection operators used in GAs. The first operator selects the pair of chromosomes for the crossover. The second one chooses the chromosomes of the next generation. The question is "On what basis chromosomes can be selected?". Several techniques have been proposed to answer this question such as ranking, random, and roulette wheel. The ranking selects the chromosomes according to their adaptation, measured by the fitness function, to have an offspring and a new generation made up of better chromosomes. Chardigny[37] explains that this technique may reduce genetic diversity because weak chromosomes will not pass their genes. The random technique, as the name suggests, involves selecting a random chromosome from the population. But still, there is a risk of losing the best genes which could lead to a mediocre population. The roulette wheel brings a balance between the previous two techniques. It highlights the best chromosomes while giving a chance to the weak ones to be selected. This technique assigns to each chromosome a probability

proportional to its adaptation before performing the selection. The roulette wheel was used in our GA to select the parents to be crossed. Nevertheless, inspired by [37], this technique was slightly modified for the selection of the chromosomes of the new generation. An aging process that determines the life span of chromosomes was integrated.

— **Crossover operators:** from two parents, the crossover operators generate two new chromosomes, each inheriting part of the parents' genes. However, since the union of the chromosome genes must correspond to all the classes of the existing application, such crossover seems inappropriate. Indeed, offsprings can end up with duplicated classes or missing ones. Therefore, the proposed crossover operator in [37] was adopted. This operator consists in keeping all the genes of one of the parents, adding a part of the genes of the second parent and then eliminating duplicates (Figure 3.12).

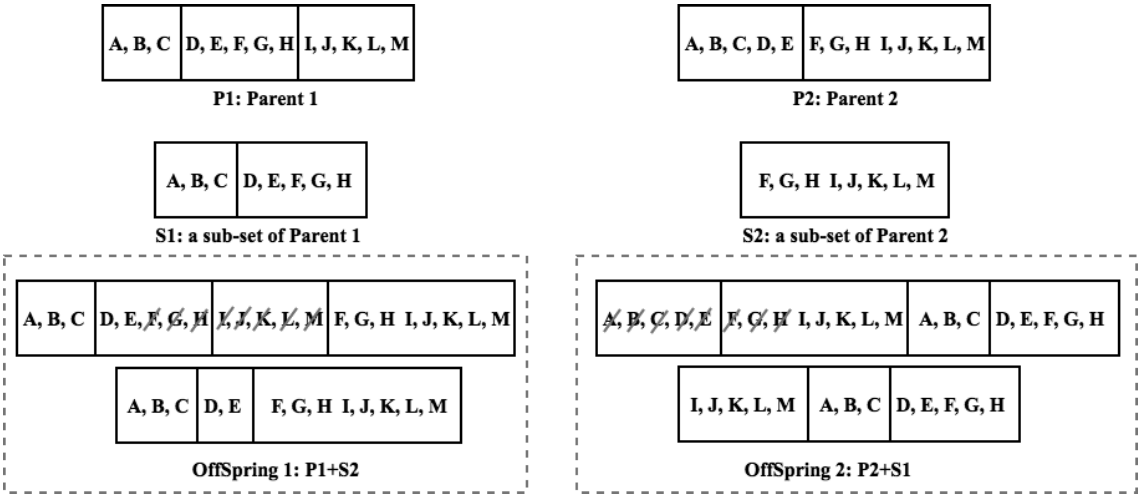

Figure 3.12 – *Example of the crossover operator*

— **Mutation operators:** these operators modify the chromosomes to introduce new genes. In the original version of GA, genes were assumed to be binary digits [88]. Thus, the mutation consists in randomly inverting a bit. In our case, since losing any gene is not acceptable (for the completeness of the system), the mutation will be either a gene fusion, separation, or both [37]. Fusion consists in grouping two genes, while separation divides the classes of a gene into two distinct genes. Figure 3.13 shows the mutation of a chromosome depending on whether it undergoes a fusion, a separation, or both.

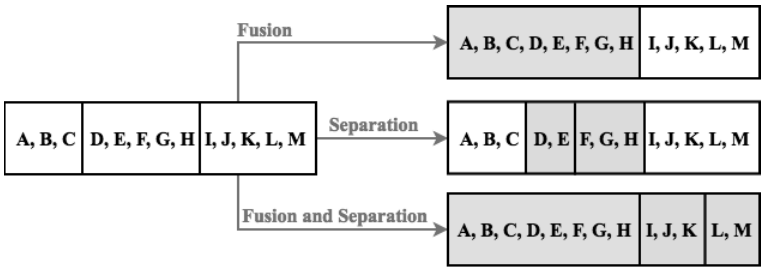

Figure 3.13 – *Example of the mutation operator*

3.5.1.3 Initial Population

The definition of the initial population is a crucial step in a GA. Depending on the diversity it presents, it can promote the success or failure of the algorithm. Hence, it is vital to ensure that chromosomes are varied, especially when performing random generation. Our initial population consists of 50 chromosomes, a number which allows widening the research space while increasing the possibilities of chromosomes diversification.

3.5.1.4 Multi-objective Genetic Algorithms

Mainly, there are two types of modeling of an optimization problem [117]: single-objective and multi-objective. The goal of a single-objective optimization is to find the solution that has the best global result. This result is calculated by combining the values of all its characteristics. The combination may follow a weighting scheme (i.e., performing a weighted sum of all the objectives).

In a multi-objective optimization, the solution is not unique. Usually, the goal of these optimizations is to find all the possible trade-offs between several objective functions. Our aim is to direct the exploration towards the solutions that present the best trade-offs between the function $FStructureBehavior$ and the function $FData$. Therefore, the multi-objective genetic algorithm proposed in [104] was chosen. This algorithm produces optimal non-dominated solutions or Pareto optimal ones, where no objective can be improved without degrading the value of one or more other objectives.

One of the particularities of the chosen algorithm is the establishment of an elitist strategy that preserves and adds to the new population the chromosomes having the best results in each objective. This strategy aims to increase the chances of spreading the best genes. At the end of its execution, the algorithm presents a set of Pareto optimal solutions to the architect, allowing him to make his choice.

3.5.2 Identification of Microservices Using a Multi-objective Genetic Algorithm

3.5.2.1 A Genetic Algorithm for Automatic Identification of Microservices

Our approach aims to identify microservices from OO source code. For that purpose, the generic algorithm proposed in [104] was adapted to our problem, as shown in Algorithm 9. This algorithm starts by initializing the population randomly (line 1). The initialization aims essentially to ensure that the population consists of various chromosomes. Then, it updates the Pareto optimal solutions (line 6) and chooses elites (lines 7 and 8) based on the computed values of *FStructureBehavior* and *FData* for each chromosome (lines 3 to 5). Once Pareto solutions and elites are determined, the next step is applying genetic operators. To avoid population explosion, they are applied only N times. Finally, the new population is generated. It contains the offsprings, the elites, and the selected chromosomes from the current population (lines 19 and 20).

Algorithm 9: Automatic identification of microservices

input : A set of OO classes $S_{Classes}$
A number of iteration I
A probability of mutation P_m
output: A set of Pareto optimal solutions $Pareto$

```
1  initialize the population P using S_Classes;
2  for i ← 1 to I do
3      for each chromosome c ∈ P do
4          Compute FStructureBehavior(c) and FData(c);
5      end
6      update Pareto;
7      E_1 ←Elite(FStructureBehavior);
8      E_2 ← Elite(FData);
9      for j ← 1 to N do
10         for each chromosome c ∈ P do
11             Compute FMicro(c);
12         end
13         (c_1, c_2) ← Select a pair of chromosomes;
14         (offspring_1, offspring_2) ←Crossover(c_1, c_2);
15         mutate(offspring_1, P_m);
16         mutate(offspring_2, P_m);
17         save offspring_1 and offspring_2 in offspring;
18     end
19     P_{i+1} ← offspring + E_1 + E_2;
20     P_{i+1} ← Select(P);
21 end
22 return Pareto;
```

3.5.2.2 Semi-automatic Identification of Microservices Based on a Genetic Algorithm

Whenever software architect recommendations are available, they are used to guide the identification process. The question is "How can they be combined with a multi-objective genetic algorithm ?". These recommendations are related to architectural aspects of the microservices such as their number, if not the range of that number, and so on. Therefore, on the one hand, they are integrated at the initialization of the population. On the other hand, they will not be affected by genetic operators. For instance, if the identification is guided by the entire set of gravity centers, even after applying genetic operators each gene should contain one gravity center. Compared to automatic identification, only the initialization is different. The remaining steps of the genetic algorithm are unchanged, except that architect recommendations will not be affected by genetic operators.

Population Initialization Based on the Knowledge of the Gravity Centers

Depending on the available gravity centers, two cases were considered:

— **Population initialization based on the entire set of gravity centers:** initializing a population based on the entire set of gravity centers aims to create an initial population consisting of *S* chromosomes (i.e., possible decomposition of the OO application). Each chromosome should contain a set of genes and each gene includes one gravity center. Algorithm 10 shows the proposed initialization.

Algorithm 10: Population initialization based on the entire set of gravity centers

input : A set of OO classes $S_{Classes}$
A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
A population size S
output: An initial population P

```
1  for i ← 1 to S do
2      Temp ← S_Rclasses;
3      Let Chromosome be a chromosome;
4      for each Class ∈ S_Gcenters do
5          add Class to Gene;
6          add n random classes of Temp to Gene ;
7          remove the n random classes from Temp;
8          add Gene to Chromosome;
9      end
10     add Chromosome to P;
11 end
12 return P;
```

This algorithm iteratively starts by creating a chromosome and its genes (lines 2 to 9). Then, the created chromosome is added to the population (line 10). Each gene consists of a gravity center (line 5) as well as a random number of the OO application classes (line 6). Therefore, the number of genes in each chromosome equals the number of gravity centers.

— **Population initialization based on a sub-set of gravity centers:** when only a sub-set of gravity centers is available, the initialization generates S chromosomes. Each one consists of a set of genes. However, not all the genes contain a gravity center, as specified in Algorithm 11. To carry out such initialization, the idea is to firstly create the genes containing gravity centers (lines 2 to 9) and then generate random ones (lines 10 to 13).

Algorithm 11: Population initialization based on a sub-set of gravity centers

input : A set of OO classes $S_{Classes}$
A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
A population size S
output: An initial population P

```
1  for i ← 1 to S do
2      Temp ← S_Rclasses;
3      Let Chromosome be a chromosome;
4      for each Class ∈ S_Gcenters do
5          add Class to Gene;
6          add n random classes of Temp to Gene ;
7          remove the n random classes from Temp;
8          add Gene to Chromosome;
9      end
10     while Temp is not empty do
11         create a random gene Random_Gene;
12         add Random_Gene to Chromosome;
13     end
14     add Chromosome to P;
15 end
16 return P;
```

Population Initialization Based on the knowledge of the Number of Microservices

— **Population initialization based on the exact number of microservices:** when the exact number of microservices Nb_{req} is provided, the goal of the initialization is to generate S chromosomes, such that each one consists of Nb_{req} genes. To create each chromosome, the classes of the OO application are partitioned randomly. Algorithm 12 shows the proposed initialization.

Algorithm 12: Population initialization based on the exact number of microservices

input : A set of OO classes $S_{Classes}$
A number of requested microservices Nb_{req}
A population size S
output: An initial population P

```
1 for i ← 1 to S do
2 |   create a random chromosome Chromosome consisting of Nb_req genes;
3 |   add Chromosome to P;
4 end
5 return P;
```

— **Population initialization based on an interval of the number of microservices:** the only difference between the previous initialization and this one is that here the number of required microservices is a random one within the provided interval. Algorithm 13 presents this initialization. To create a new chromosome, the first step is choosing a random number Nb_{rand} within the specified interval (line 2), followed by generating Nb_{rand} genes for this chromosome (line 3).

Algorithm 13: Population initialization based on an interval of the number of microservices

input : A set of OO classes $S_{Classes}$
A minimum number of microservices Nb_{min}
A maximum number of microservices Nb_{max}
A population size S
output: An initial population P

```
1 for i ← 1 to S do
2 |   let Nb_rand be a random number within [Nb_min, Nb_max];
3 |   create a random chromosome Chromosome consisting of Nb_rand genes;
4 |   add Chromosome to P;
5 end
6 return P;
```

Population Initialization Based on the Exact Number of Microservices and a Sub-set of Gravity Centers

This initialization combines two recommendations: the exact number of microservices Nb_{req} and a sub-set of gravity centers. Therefore, on the one hand, any chromosome should consist of Nb_{req} genes. On the other hand, each gene contains at most one gravity center (Algorithm 14).

Algorithm 14: Population initialization based on the exact number of microservices and a sub-set of gravity centers

input : A set of gravity centers $S_{Gcenters}$
A set of remaining OO classes $S_{Rclasses}$
A number of requested microservices Nb_{req}
A population size S
output: An initial population P

```
1 for i ← 1 to S do
2 |   create a random chromosome Chromosome consisting of Nb_req genes such that
  |     no gene contain more than one class of S_Gcenters;
3 |   add Chromosome to P;
4 end
5 return P;
```

3.6 Conclusion

The main contribution of the work presented in this chapter is the proposal of an approach for the identification of microservices from OO source code. The proposed approach is based mainly on two types of information: the source code information and the knowledge, often partial, of the architect concerning the system to migrate. On the one hand, the source code information includes relationships between its elements as well as their relationships with the persistent data manipulated in this code. In fact, source code information is used by a quality function to evaluate the relevance of a candidate microservice. This function was defined based on the analysis of microservice characteristics. On the other hand, architect recommendations are related, mainly, to the use of the applications (e.g., how many microservices, which class is the center of a microservice, etc.). Our approach proposes to identify microservices using a hierarchical clustering algorithm or a multi-objective genetic one. Moreover, it can be automatic or semi-automatic, based on whether software architect recommendations guide it or not.

IV

Task-based Migration To Microservices: An Approach Based on Workflow Extraction from Source Code

4.1 Introduction

Microservices can be identified relying on two approaches, namely structure-based identification and task-based one. In Chapter 3, the structure-based approach has been proposed. This chapter presents our contribution to the task-based identification, which is the extraction of a workflow from OO source code to ensure that existing task-based identification approaches can be applied, even when the only available artifact is the source code.

The extraction of a workflow from OO source code requires the ability to map OO concepts into workflow ones. For instance, specifying what is the mapping of the concept task compared to the OO concepts is necessary. Once such a mapping is established, workflow constituents (i.e., tasks, control flow, and data flow) are recovered from the OO source code relying on it.

This chapter is organized as follows. Section 4.2, firstly, presents a possible mapping from OO concepts to workflow ones. Then, it introduces the proposed extraction process. Section 4.3 presents a solution for task identification. Sections 4.4 and 4.5 present respectively control as well as data flows recovery solutions. Finlay, Section 4.6 concludes this chapter.

4.2 Approach Principals

4.2.1 From Object-Oriented Architectural Style to Workflow-based one: The Mapping Model

A task is the basic unit of composition of a workflow. It is the smallest execution unit, representing an execution stage in a whole application. Based on this definition, we consider that a method can be mapped to a task in a workflow (Figure 4.1). In particular, we assume that a method that contains only assignment statements or invokes only methods provided by a standard library is mapped to a primitive task, whereas a method that includes a sequence of methods invocations and control statements is mapped to a composite task. For methods including both methods invocations, control statements, and other statements, the source code has to be refactored to wrap these later in a method. In fact, a workflow, as seen in Figure 4.1, does not include assignment statements as a unit of composition.

A task can define input and output data, where input data represent data required for executing the task and output data, the produced ones. Since a task corresponds to an OO method, task inputs represent data needed to execute the corresponding method, while task outputs are the data generated from this execution. To be executed, a method requires a receiving object and its input parameters. Once executed, the method's produced data are its modified inputs (i.e., receiving object and/or input parameters), and/or its output parameter (i.e., returned value). Hence, each method's produced data that represents a modified input corresponds to two data in a workflow, an input and an output data (Figure 4.1).

In a workflow, the execution order of tasks is expressed using different control constructs such as sequence, conditional branches and loops. In OO style, the

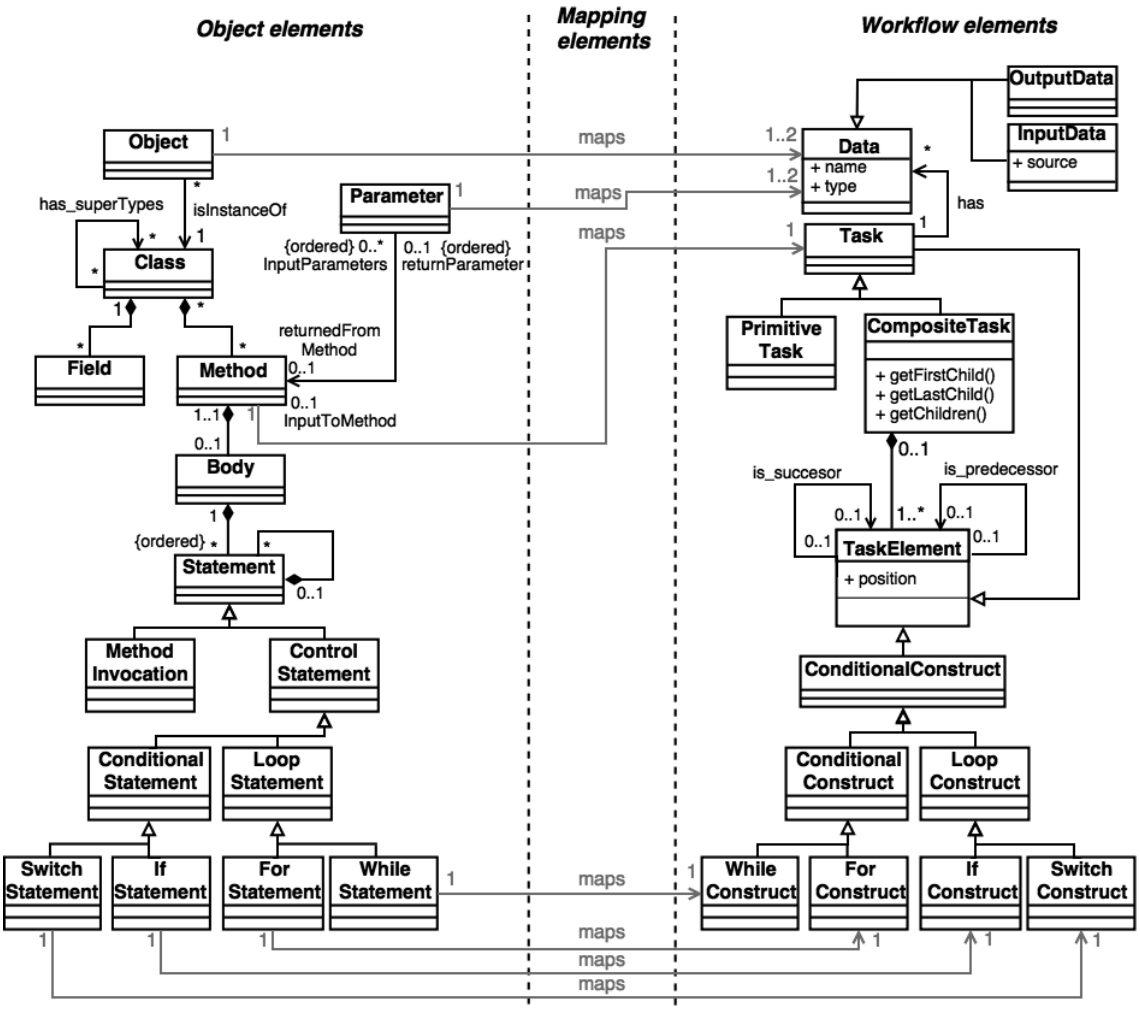

Figure 4.1 – *From OO elements to workflow ones: the mapping model.*

execution order of methods invocations, which corresponds to the execution of tasks, depends on control statements (e.g., if statement, while statement, etc.). Hence, we consider that a control statement can be mapped to a control construct. Thus, the input data of a control construct corresponds to data manipulated in the corresponding control statement (i.e., in the condition and the body of the control statement), while control construct outputs are the data defined in the control statement and used in the following statements.

The explained mapping between OO concepts and workflow ones is illustrated in Figure 4.1.

4.2.2 Extraction Process

Our approach aims to recover a workflow based on static analysis of OO source code. For that purpose, an extraction process has been proposed (Figure 4.2). It consists of two steps:

— ***Step 1:*** the aim of this step is recovering a model of the OO source code. It involves identifying the application structural elements (e.g., classes, methods, etc.) and their relationships (e.g., method calls, class inheritances, etc.) by analyzing the existing source code.

— ***Step 2:*** the second step consists of transforming the OO model into a workflow-based one relying on the proposed mapping model. It starts by identifying primitive and composite tasks, as well as their respective input and output data, and then recovering the control flow, and the data flow associated to these tasks so that the extracted workflow represents the behavior of the OO application.

In order to realize this process, the following questions need to be answered:

— What are the tasks that reflect the workflow corresponding to the existing OO application? Answering this question requires identifying, at the instance level, a matching between task entities and OO methods invoked on object instances.

— What is the control flow to be specified between the identified tasks to represent the behavior of the existing OO application? Answering this question requires, along with others, rendering explicit the implicit control flow due to OO features (e.g., polymorphism and dynamic binding, etc.).

— What is the data flow to be associated with the identified tasks and control flow? The aim is defining for each task its input and output data so that the recovered workflow specifies the correct data dependencies between

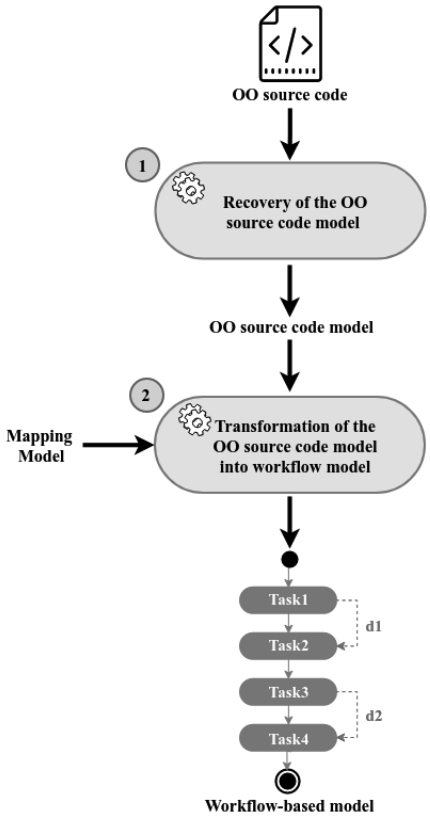

Figure 4.2 – *Process of extracting workflows from OO source code*

tasks. This is mainly identifying the flow of objects associated with the tasks already identified.

An overview of task identification, control flow recovery, and data flow recovery enabling to explain how these questions are answered will be presented and illustrated (Figure 4.3).

— **Task identification:** it starts by applying extract method refactoring on methods containing a combination of method calls, control statements, and other statements. As explained earlier, tasks correspond to methods, and their type (i.e., primitive or composite) depends on whether they call other methods or not. Therefore, the call graph of the refactored source code is built and analyzed to identify them. To finalize the identification of tasks, their inputs and outputs are specified based on the DEF and USE sets of their corresponding methods. The DEF (resp., USE) set contains parameters and attributes defined (resp., used) by these methods.

— **Control flow recovery:** control flow recovery aims to build a Control Flow Graph (CFG) for each composite task. To construct this graph statically, dynamically dispatched calls are refactored into nested if-else statements.

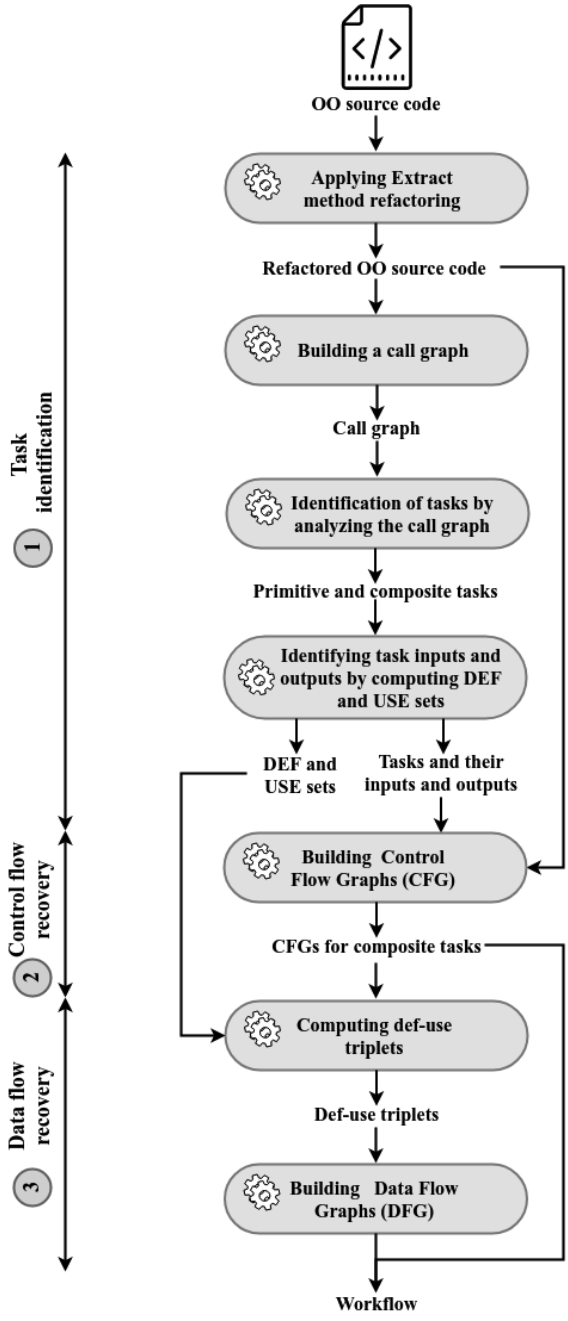

Figure 4.3 – *Overview of task identification, control flow recovery, and data flow recovery*

— **Data flow recovery:** data flow recovery aims to build a Data Flow Graph (DFG) for each composite task. For that purpose, the def-use triplets for the corresponding methods to composite tasks are firstly computed and then used to construct DFGs. Each triplet (var, def, use) specifies the name of a

variable var, the line number at which this variable is defined def, and the line number at which this definition is used use.

4.3 Identifying Tasks from OO Source Code

Our mapping model between OO concepts and workflow ones establishes a unique mapping for workflow tasks. A task is mapped to an OO method (Figure 4.1). Therefore, relying on code refactoring, and more precisely extract method refactoring, all statements that do not represent method invocations in the OO source code are transformed into method invocations. Once this refactoring is done, among all the methods in the OO source code, those that correspond to primitive tasks and those that correspond to composite ones are determined. Finally, the input and output data for each one of them are specified.

4.3.1 Extract Method Refactoring

The refactoring of OO source code consists of extracting each sequence of statements delimited by user method invocations as a new method and replacing this sequence with an invocation of the newly extracted method. Note that only statements belonging to the same block can be extracted to guarantee the syntactical correctness of the new method.

Figure 4.4 shows an example of extract method refactoring. In this example, the statements delimited by method invocations *s.initializeInputs()* and *s.updateInputs(internalInput)* cannot be extracted as a new method because they do not belong to the same block. However, it is possible to divide this sequence into two fragments based on whether the statements belong to the same bloc or not. Each fragment is extracted as a new method (method *m1* and method *m2*).

Once the statements to be extracted are specified, the extraction starts. Variables acceded (i.e., defined or used) but not declared by these statements (i.e., the variables declaration statements do not belong to these statements) should be passed in as input parameters of the new method. Whereas, variables defined by these statements and acceded by following ones should be passed out as its output parameters. Note that the definition of a variable means that its value is modified (i.e., writing access), while its usage implies that its value is read (i.e., reading access).

In the example shown in Figure 4.4, the statement extracted as the method *m2* uses the variables *internalInput* and *step*. Therefore, these variables are passed in

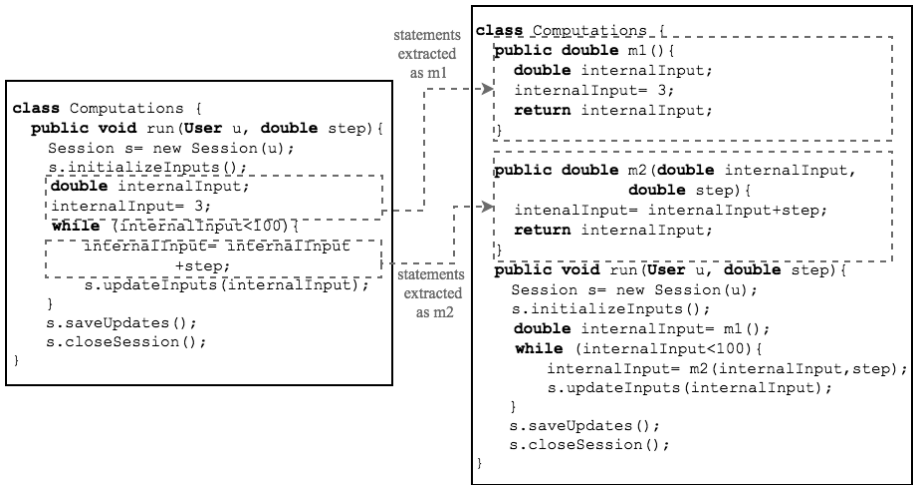

Figure 4.4 – *Example of extract method refactoring*

as input parameters of *m2*. Moreover, the variable *internalInput* is defined by the statement to be extracted as *m2* and used in the method invocation *s.updateInputs-(internalInput)*, and thus it is an output parameter of *m2*.

Some OO languages (e.g., Java, etc.) imposes that a method can have at most one output parameter (i.e., returned value). For this reason, if a sequence of statements to be extracted has more than one output value, this sequence should be divided into multiple fragments. Each one of them is extracted as a new method and return at most one output parameter. It is noteworthy that this code fragmentation and method extraction do not re-order code statements, which excludes the possibility of breaking down program semantics (i.e., the program's behavior is preserved).

4.3.2 Identifying Task Based on Analyzing the OO Application Call Graph

As explained earlier in Section 4.2.1, there are two types of tasks in a workflow: primitive and composite ones. Since a task corresponds to a method in OO source code, a method that does not contains calls to others is considered as a primitive task. Otherwise, it is a composite one. Therefore, the identification of primitive and composite tasks is based on the analysis of the OO application's call graph.

4.3.2.1 Choosing a Call Graph Construction Algorithm

A call graph is an artifact that specifies for each caller method its callees. It can be built by statically analyzing the source code. In OO languages with dynamic dispatch, the crucial step in building a call graph is determining a conservative approximation of the set of methods that can be invoked by a virtual method call (i.e., a dynamically dispatched call) [132].

In literature, several algorithms [49, 20, 21, 132, 129] have been proposed to construct call graphs based on static analysis of the source code. Their objective is to build accurate graphs (i.e., consisting of the fewest nodes and edges possible) while being sound (i.e., containing all the edges that may occur at run-time). The main difference between most of these algorithms is the used information to approximate the run-time types of a receiver [132]. Intuitively, algorithms that use more information build a more accurate call graph, but they need more time and space [132], which might affect their scalability.

Several well-known algorithms will firstly be presented briefly (Appendix A for more details), and then one of them will be chosen. These algorithms are the following:

— **Reachability Analysis (RA):** RA is a simple algorithm for constructing a call graph. It only considers the name of a method. A slightly more advanced version of this algorithm takes into account method signatures instead of their names.

— **Class Hierarchy Analysis (CHA):** the call graph constructed using RA can be improved by considering the class hierarchy. For that purpose, CHA [49] was proposed. This analysis uses the static type of a receiver of a virtual call, combined with the class hierarchy, to determine the methods that can be invoked at run-time.

— **Rapid Type Analysis (RTA):** RTA [20, 21] is an improvement of CHA by taking into account class instantiation information. A possible type of a receiver can only be a class that has been instantiated in the OO application via a *new*.

— **XTA analysis and its variants:** XTA [132] is an improvement of RTA. The idea is that by giving methods and fields a more precise local view of the types of objects available, virtual calls may be resolved more precisely. By associating a distinct set to each class, method and/or field, different variants of XTA can be obtained (i.e., Class Type Analysis (CTA), Method Type Analysis (MTA) and Field Type Analysis (CTA)).

— **Variable Type Analysis (VTA):** another enhancement of RTA is VTA [129]. Instead of collecting class instantiations from the OO application and use

them to prune the call graph, VTA relies on the types of objects that can reach each variable (i.e., the instantiations that might be assigned to this variable).

In [132], a schematic overview showing the relationship between these algorithms has been proposed. Figure 4.5 shows a reproduction of this overview with a simple modification (i.e., adding VTA). All the presented algorithms have been shown to scale well [132, 129].

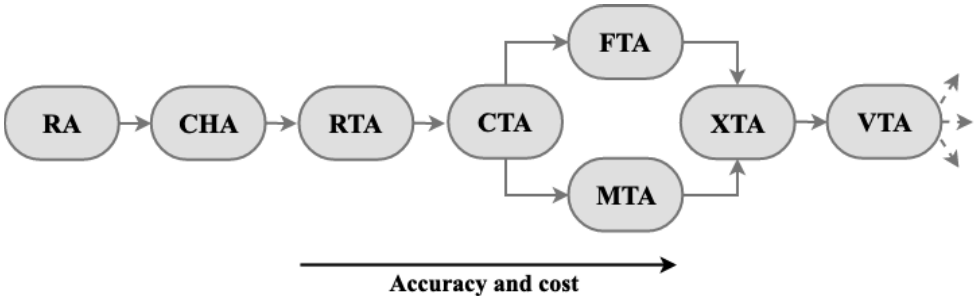

Figure 4.5 – *Schematic overview of the presented call graph construction algorithms and their relationship.*

In our approach, we chose to use XTA. Firstly, it is cheap and produces an accurate call graph compared to other algorithms. Secondly, it is simple, easy to implement, and scales well. Finally, unlike VTA, it does not require a 3-address representation to simplify the analysis.

4.3.2.2 Identifying Tasks

Once the call graph is built, its leaves are mapped to primitive tasks, while the remaining nodes are mapped to composite ones. Nevertheless, when analyzing the call graph to identify tasks, there is a particular case that needs to be handled. This case concerns direct or indirect recursive calls. Since workflows do not always support recursive transitions between tasks (i.e., the ability of a task to invoke itself directly or indirectly during its execution) [116], recursive calls between methods are transformed as follows: a method *M* in a directed cycle is mapped to a primitive task if all its invoked methods belong to this cycle (method *Foo.incX* in Listing 4.1). Otherwise, this method is mapped to a composite task (method *Foo.setX* and *Foo.performComputations*).

Figure 4.6 represents the call graph built from the source code shown in Listing 4.1. The methods *Main.main*, *Bar.m*, *Foo.setX*, and *Foo.performComputations* cor-

respond to composite tasks, whereas the methods *Foo.initializeX*, *Foo.getX*, *Foo.incX*, *Foo.printX* and *Foo.multiplyX* are mapped to primitive ones.

Listing 4.1 – *Classes Foo, Bar and Main*

```
1  class Foo {
2          int x;
3          void setX(int y, boolean isDifferent ){
4                  if (isDifferent)
5                      initializeX(y);
6                  else
7                      performComputations(y);}
8          void initializeX(int y) {
9                  x=y;}
10         void performComputations(int y){
11                 int z= multiplyX(y);
12                 incX(z);}
13         int multiplyX(int y) {
14                 return 2*y;}
15         int incX(int y){
16                 if (x!=y)
17                     setX(y, true);
18                 else
19                     setX(y, false);
20                 return x;}
21         int getX(){
22                 return x;}
23         void printX(int y) {
24                 System.out.println(y);}}
25 class Bar{
26         Foo foo= new Foo();
27         void m(){
28                 foo.initializeX(1);
29                 int x= foo.getX();
30                 while (x<20){
31                     x=foo.incX(x);
32                     foo.printX(x);}}}
33 class Main {
34         static Bar bar= new Bar();
35         public static void main(String[]args){
36                 bar.m();}}
```

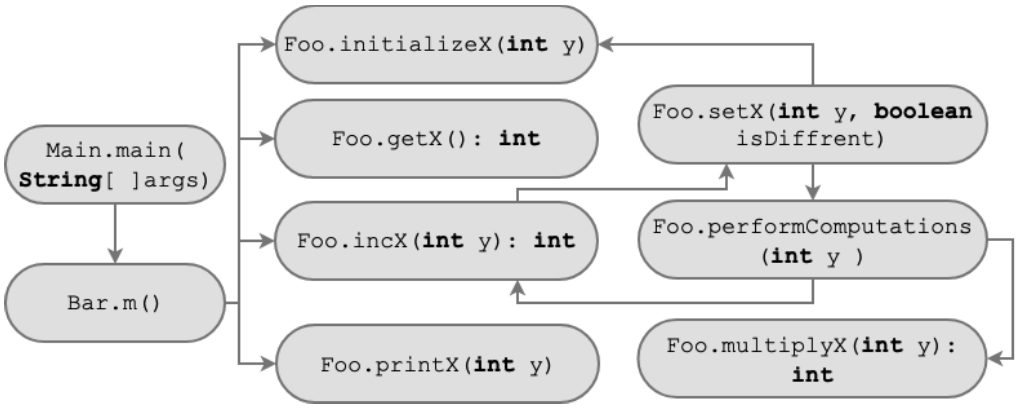

Figure 4.6 – *Call graph built from the source code shown in Listing 4.1*

4.3.3 Identifying Tasks Inputs and Outputs

Each task in a workflow, primitive or composite, has input and output data. Therefore, once the tasks are identified from OO source code, their input and output data need to be specified. As explained in Section 4.2.1, task inputs are the parameters and the receiving object of the corresponding method. While its outputs correspond to the modified inputs and the returned value by this method. Note that task outputs include its modified inputs because this task generates their new values.

To identify the inputs that can be modified by a method, and thus the corresponding task outputs, for each method *M* two sets are computed: *DEF* and *USE* sets. These sets were inspired by [43]. The *DEF* (resp., *USE*) set contains parameters and attributes defined (resp., used) by *M*. In other words, parameters and attributes that their values are modified (resp., read) by *M*. A task input *INDATA* is considered as modified if either 1) *INDATA* is the receiving object of *M* and at least one of its attributes $\in$ *DEF(M)* or 2) *INDATA* $\in$ *DEF(M)*.

For instance, the inputs of the corresponding task to the method *initializeX* of the class *Foo*, shown in Listing 4.1, are the receiving object and the parameter *y*, whereas the output of this task is the receiving object because its attribute *x* is defined in the method *initializeX* (in Line *9*). However, for the corresponding task to the method *getX* of the class *Foo*, the receiving object is only input since its value is not modified in this method.

4.3.3.1 Computing DEF and USE sets

In our approach, we consider that assignment on a variable of a primitive type (i.e., a type which is not a class) as a *DEF* operation. All other operations on primitive variables are considered as *USE* ones [99]. Nevertheless, when dealing with objects, we assume that operations are *DEF* ones in three cases: 1) this operation defines at least one attribute of the object 2) it is a constructor invocation or 3) it is a call of a method that modifies this object (i.e., modifies at least one attribute of this object) [43]. Otherwise, these operations are of *USE* category.

Based on these cases, determining whether an input *INDATA* of a method *M* (i.e., a parameter or the receiving object) is in the *DEF* or to *USE* sets depends on the *DEF* and *USE* sets of the called methods by *M*. An input *INDATA* of *M* is considered as defined (resp., used) if it is defined (resp., used) by either 1) a statement of *M* which is not a method invocation (e.g., assignment, etc.) or 2) at least one of the methods called by *M*. More precisely, an input *INDATA* is considered as defined (resp., used) in a called method *CalledM* by *M* in two cases.

— *Case 1: INDATA* is the receiving object of the invocation of *CalledM*, and at least one of the attribute of *INDATA* is in the *DEF*(resp., *USE*) set of *CalledM*. For instance, the receiving object *foo* of the invocation of the method *initilizeX* in Line *28* (Listing 4.1) is considered as defined since the method *initializeX* defines the attribute *x* of the receiving object in Line *9*.

— *Case 2: INDATA* is passed as a parameter in the invocation of *CalledM*, and its corresponding formal parameter belongs to the *DEF*(resp., *USE*) set of *CalledM*. In the example shown in Listing 4.1, the input *y* passed as parameter in the invocation of the method *initializeX* in Line *28* is considered as used because its corresponding formal parameter *y* is used in the method *initializeX* (Line *9*).

The above constraints related to computing *DEF* sets can be formalized as follows:

$$
\begin{cases}
********** & *** \\
- INDATA\ is\ defined\ in\ M & (1) \\
- \exists\ stat \in \{Statements(M) - MethodCalls(M)\} & (2) \\
- INDATA\ is\ defined\ in\ stat & (3) \\
- \exists\ call \in MethodCalls(M) & (4) \\
- ReceivingObj(call) = INDATA & (5) \\
- \exists\ attribute \in AttributeOf(INDATA) & (6) \\
- attribute \in DEF(CorrespondingMethod(call)) & (7) \\
- INDATA \in ActualParameter(call) & (8) \\
- FormalParameter(INDATA) \in DEF(CorrespondingMethod(call)) & (9) \\
********** & *** \\
(1) \Rightarrow ((2) \wedge (3)) \vee ((4) \wedge (((5) \wedge (6) \wedge (7)) \vee ((8) \wedge (9))) & \\
********** & ***
\end{cases}
$$

Where:

— $Statements(M)$ denotes the set of statements of M.

— $MethodCalls(M)$ represents the set of method calls in M.

— $ReceivingObj(call)$ specifies the receiving object of a method call.

— $AttributeOf(INDATA)$ denotes the set of attributes of $INDATA$.

— $ActualParameter(call)$ determines the set of actual parameters in a $call$.

— $FormalParameter(INDATA)$ indicates the corresponding formal parameter of an actual parameter $INDATA$.

— $CorrespondingMethod(call)$ represents the called method in $call$.

Note that by replacing the *DEF* set with *USE* set in the formula presented above, it specifies when an input $INDATA$ is considered as used.

Based on the above constraints, it is clear that determining *DEF* and *USE* sets of any method *M* depends on the sets of its called methods. For this reason, the computation of *DEF* and *USE* sets requires an analysis order of methods. For instance, to compute *DEF* and *USE* sets of the method *m* (Listing 4.1), *DEF* and *USE* sets of the called methods by *m* on the attribute *foo* (*initializeX*, *getX*, *incX* and *printX*) are needed to check whether *foo* is defined and/or used.

The built call graph allows the definition of a topological total order of its nodes. Analyzing methods according to this order ensures that a called method is always analyzed before its callers. In the total order, the first methods are the ones that do not contain any others. These methods correspond to the leaves of the graph.

4.3.3.2 Handling Call Graph Cycles

In the presence of direct or indirect recursion, the call graph contains cycles. Therefore, it is not possible to define an analysis order. Replacing each cycle in the graph with a representative node allows solving this problem (i.e., allows the definition of a total order). Once the order is defined, the call graph nodes are then analyzed following this order. If a node represents a method, then *DEF* and *USE* sets are computed relying on the former constraints. If the node is a representative, *DEF* and *USE* sets of each method in the cycle represented by this node are calculated, firstly, without taking into account calls to the methods belonging to the cycle, and then these sets are re-computed while considering the calls between the methods of the cycle.

For example, computing *DEF* and *USE* sets of the methods shown in Listing 4.1, requires replacing the cycle containing methods *setX*, *performComputations* and *incX* with a representative node *Cycle1* (Figure 4.7). Once the cycle is replaced, the nodes can be ordered as follows: 1) Foo.initializeX, 2) Foo.getX, 3) Foo.multiplyX, 4) Foo.printX 3) Cycle1, 4) Bar.m and 5) Main.main. Table 4.1 shows *DEF* and *USE* sets computed for these methods.

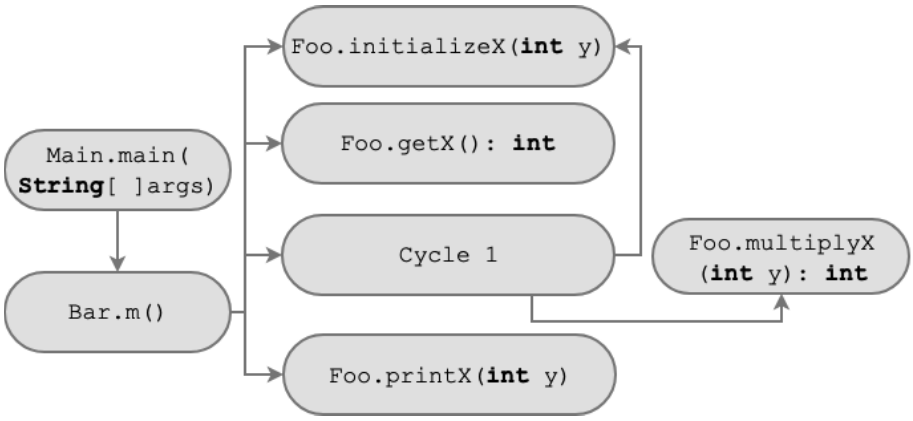

Figure 4.7 – *Acyclic call graph*

Table 4.1 – *DEF/USE sets of the methods shown in Listing 4.1*

Method	DEF set	USE set
initializeX	{x}	{y}
getX	∅	{x}
multiplyX	∅	{y}
printX	∅	{y}
setX	{x}	{x, y, isDifferent}
performComputations	{x}	{x,y}
incX	{x}	{x, y}
m	{foo}	{foo}
main	{bar}	{bar}

4.4 Control Flow Recovery

A workflow can be viewed as an application consisting of a set of tasks, with possible dependencies specified relying on control and data flows. Therefore, once the tasks are identified, the corresponding control flow needs to be recovered. This later specifies the execution order of tasks. When dealing with composite ones, their control flow describes the execution order of their enclosed tasks.

In our approach, a control flow is represented as a graph, named Control Flow Graph (CFG). A CFG consists of a set of nodes and a set of edges. Each node represents either a method call, a predicate, or a control. A predicate specifies a condition utilized in a control statement, for instance, if statement. Note that CFG nodes representing a method call or a predicate are labeled relying on their line number in the source code. CFG edges specify the execution order of method calls and evaluation of predicates.

To build a CFG, the statements, which represents the body of the corresponding method to the composite task, are traversed and the graph is constructed incrementally. For example, the CFG corresponding to the composite task which is the mapping of the method *m* of the class *Foo*, shown in Listing 4.1, is illustrated in Figure 4.8.

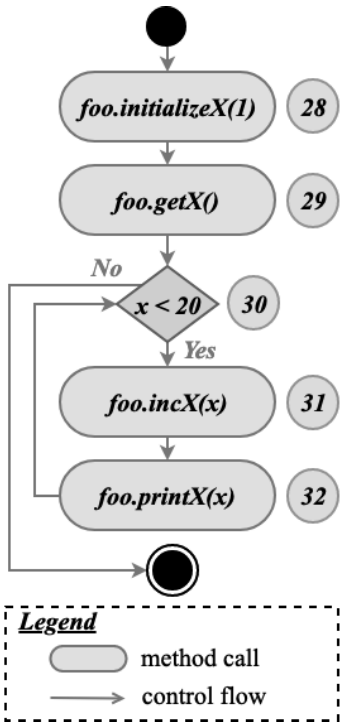

Figure 4.8 – *The CFG of the composite task corresponding the method m of the class Foo shown in Listing 4.1*

As explained earlier, in the CFG, a node can represent a method call. Nevertheless, if this call is dynamically dispatched, it is not possible to resolve statically (i.e., at compile time) the exact method to invoke. Hence, to enable building a CFG statically, our idea is to refactor each dynamically dispatched call by replacing it with nested if-else statements. In these statements, conditions represent the possible run-time types of the receiving object of the call, whereas the branches are the different implementations of the called method in each possible receiving object type.

An example of a CFG recovered in the presence of dynamically dispatched calls is shown in Figure 4.9. The method *m* of the class *Bar* calls the method *printX* of the class *Foo*, the class *FooExp* or the class *FooPro*. Therefore, the corresponding CFG contains a path for each possible run-time type of the receiver (i.e., *Foo*, *FooExp*, and *FooPro*).

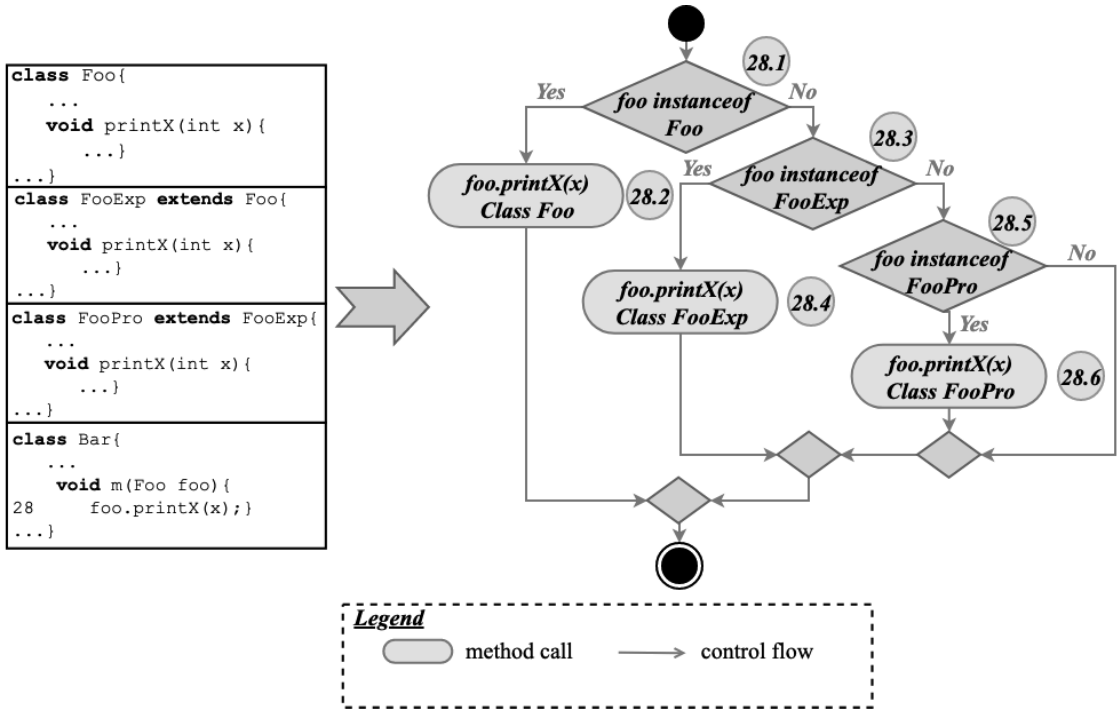

Figure 4.9 – *Example of a CFG recovered in the presence of dynamically dispatched calls.*

4.5 Data Flow Recovery

In addition to the identification of tasks and control flow, the construction of a workflow also requires the recovery of data flow, which specifies the dependency links between the data of the identified tasks (i.e., which output data of a task represents an input data of another one).

4.5.1 Data Flow Graph Construction

In our approach, a data flow is represented as a graph, named Data Flow Graph (DFG). A DFG has the same nodes as a CFG. Nevertheless, an edge between two nodes N_i and N_j labeled using the name of the variable v means that v is defined in N_i and used in N_j.

For instance, in the example shown in Figure 4.10, the edge between the nodes *foo.initializeX(1)* and *foo.getX()* labeled with *foo* specifies that the variable *foo* is defined in the node *foo.initializeX(1)* and used in the node *foo.getX()*.

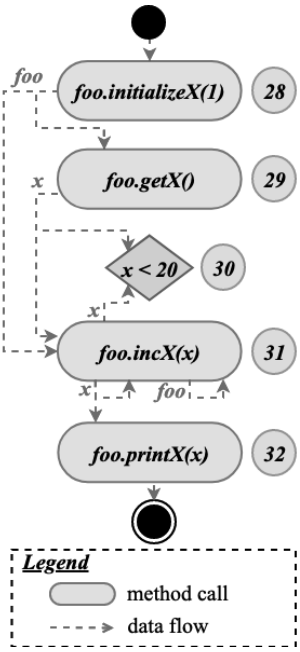

Figure 4.10 – *The DFG corresponding to the task mapped to the method m of the class Foo shown in Listing 4.1*

As explained earlier, a composite task encloses other primitive and composite ones. Hence, a data flow is recovered for each composite task to specify data dependencies between its enclosed tasks.

To build a DFG for a composite task, def-use triplets for the corresponding method to this task are computed. Each triplet *(var, def, use)* specifies the name of a variable *var*, the line number at which this variable is defined *def* and the line number at which this definition is used *use*. As explained in section 4, CFG nodes are labeled relying on the corresponding line numbers in the source code. Therefore, a data flow edge is created between two nodes denoted by *k* and *l* only if the def-use triplets of the corresponding method include the triplet *(v, k, l)*.

For example, in the *DFG* represented in Figure 4.10, an edge is created between the nodes denoted by *31* and *32* because the def-use triplets of the method *m* include the triplet *(foo,31,32)*. In the rest of this section, the process of computing def-use triplets will be explained.

4.5.2 Computing Def-Use Triplets

The process of computing def-use triplets consists of three steps. The first step aims to compute *VarUsed* sets. Each set specifies the variables used in a CFG node. The goal of the second step is to determine *ReachDef* sets. Each set specifies

the definitions that reach a node of the CFG. A definition of a variable v in a node N_i, denoted *(v,N_i)*, reaches a node N_j if there is a path in the CFG between N_i and N_j without a redefinition of v. In the third step, def-use triplets are computed relying on *VarUsed* and *ReachDef* sets. The rest of this section explains in details these steps.

Step1: Computing *VarUsed* sets: computing the *VarUsed* sets requires determining *USE* sets for all the methods in the source code (Section 4.3.3). Once these sets are determined, for each node in the CFG representing a method call, the receiving object is used (i.e., included in the *VarUsed* set) if the *USE* set of the called method contains at least one of its attributes. An effective parameter is used if the *USE* set of the invoked method includes its corresponding formal parameter. The *VarUsed* of a predicate node contains variables used in the corresponding expression.

Step2: Computing *ReachDef* sets: determining reaching definitions requires computing *DEF* sets for all the methods in the source code (Section 4.3.3). When the *DEF* sets are computed, the produced definitions by each node in the *CFG* are specified. For a node that represents a method call, the receiving object is considered as defined (i.e., included in the produced definitions) if the *DEF* set of the called method contains at least one of its attributes. An effective parameter is considered as defined if the *DEF* set of the invoked method includes its corresponding formal parameter.

Once the definitions produced by each node are specified, reaching definition are determined using the following propagation algorithm proposed by Aho et al. [9] (Algorithm 15). In this algorithm, each node *N* of the *CFG* stores the incoming and outgoing definitions respectively inside the sets *ReachDef(N)* and *ReachOut(N)*, which are initially empty. Moreover, each node *N* generates definitions contained in the *Gen(N)* set, and prevents the elements in the *kill(N)* set from being further propagated after the node *N*. Incoming definitions for a node *N* (i.e., *ReachDef(N)*) are obtained from its predecessors as the union of the respective *ReachOut* sets (forward propagation).

Step3: Computing def-use triplets: for each CFG node *N*, if a variable $v \in$ *VarUsed (N)* and *(v,def)* $\in$ *ReachDef(N)*, then a triplet *(v,def,N)* is constructed.

For example, using *VarUsed* and *ReachDef* sets (Table 4.2) computed for each node of the CFG shown in Figure 4.8, def-use triplets constructed are *(foo,28,29)*, *(foo,28, 31)*, *(foo,31,31)*, *(x,29,30)*,*(x,31,30)*, *(x,29,31)*, *(x,31,31)* and *(x, 31,32)*.

Algorithm 15: Compute reaching definitions

input : Control flow graph CFG
Gen and *Kill* sets for each node of the CFG
output: ReachDef and ReachOut sets for each node of the CFG

```
for each node n of CFG do
    ReachDef(N) ← ∅;
    ReachOut(N) ← ∅;
end
while any ReachDef(N) or ReachOut(N) changes do
    for each node N of CFG do
        ReachDef(N) ← ∪_{P∈predecessor(N)} ReachOut(P);
        ReachOut(N) ← Gen(N) ∪ (ReachDef(N) − Kill(N));
    end
end
```

Table 4.2 – *VarUsed and ReachDef computed for each node of the CFG shown in Figure 4.8*

Node	*VarUsed*	*ReachDef*
28	∅	∅
29	{foo}	{(foo, 28)}
30	{x}	{(foo, 28), (foo,31), (x,29), (x,31)}
31	{x,foo}	{(foo, 28), (foo,31), (x,29), (x,31)}
32	{x}	{(foo, 31), (x,31)}

4.6 Conclusion

The main contributions of the work presented in this chapter is an extraction approach aiming to recover a workflow from OO source code. To achieve that aim, firstly a mapping model between OO concepts and workflow ones has been defined. Then, to identify this mapping from OO source code, an extraction process consisting of three steps has been proposed. Each step recovers a workflow constituent (i.e., tasks, control flow, and data flow). It is noteworthy that the extracted workflow can be used to identify microservices or as documentation, due to its hierarchal structure.

V

Experimentations and Validations

This chapter presents the conducted experiments on case studies to validate our approaches. Firstly, Section 5.1 presents the carried out experiments to validate qualitatively and quantitatively our identification approach of microservices from OO applications. Qualitative evaluation is performed based on three case studies of different sizes, from small to relatively large. The quantitative assessment is carried out using two case studies, and the obtained results are compared with those produced during the evaluation of two state-of-the-art approaches

among the well-known ones. They have been proposed in [100] and [76]. Secondly, Section 5.2 presents the validation of our extraction approach of workflows from OO applications. Finally, Section 5.3 concludes this chapter.

5.1 Validating the Identification of Microservices from OO Applications

In this section, we present the research questions that we have attempted to answer empirically in order to validate our microservice identification approach. We also present the experimental protocols of the two types of experiments carried out to answer the research questions. These two types of experiments are respectively related to two different evaluation methods. The first one allows a qualitative assessment of the identified microservices by our approach. The second one allows a quantitative evaluation, via certain metrics measuring the functional independence of the recovered microservices. We compare the obtained measurement values of these metrics with those obtained during the evaluation of two state-of-the-art approaches among the well-known ones. Once the results of both evaluations are produced, we use them to answer the research questions. In this section, we also present our analysis of the internal and external threats to validity with respect to the conducted experiments.

5.1.1 Research Questions

To validate our proposal, we conducted experiments to answer the following research questions:

— **RQ1:** does the proposed quality function produce an adequate decomposition of an OO application into microservices?
Our approach partitions an OO application into microservices based on the proposed quality function and software architect recommendations, when available. This question aims to check whether the defined function enables obtaining relevant microservices without considering the recommendations of a software architect.

— **RQ2:** is the definition of the quality function, without considering data autonomy, adequate?
The goal behind this research question is to check whether the assessment of the characteristics "focused on one function" and "structural and behavioral autonomy" produce appropriate microservices.

— **RQ3:** does the evaluation of data autonomy characteristic enhance the quality of microservices?
This question aims to check whether the function *FData* related to the evaluation of data autonomy characteristic allows improving the quality of the identified microservices compared to those identified only based on the assessment of "focused on one function" and "structural and behavioral autonomy" characteristics.
— **RQ4:** does the use of software architect recommendations enhance the identification results?
The goal behind this research question is to check whether software architect recommendations guide our approach to produce better results.
— **RQ5:** what are the software architect recommendations that generate the best decomposition of an OO application into microservices?
Since our approach uses several recommendations, this question aims to determine the ones that produce the best results.

5.1.2 Experimental Protocol

Our experiments conducted to answer the previous research questions are based on a prototype plug-in that we developed in Java. It carries out the identification process defined in our approach. This section presents the experimental protocols followed to answer these questions based on qualitative, and quantitative evaluation of the identified microservices using our plug-in.

5.1.2.1 Experimental Protocol for the Qualitative Evaluation of Microservices

In order to answer the **RQ1**, we used our plug-in to partition three Java applications, that will be presented in Section 5.1.3.1. Since the goal of the **RQ1** is to evaluate the correctness of the proposed quality function, we set the plug-in to apply the clustering algorithm that does not take as inputs any software architect recommendations (i.e., fully automatic). Then, we compared the produced microservices with those identified manually. The manual identification was based on source code analysis and known features of these applications. It is noteworthy that to avoid biasing the results, we firstly partitioned these applications manually. After that, we carried out the identification using our plug-in.

The protocol for answering the **RQ2** is similar to the one used to answer the **RQ1** with only one difference: we set our plug-in to identify microservices based on the sub-function *FStructureBehavior* related only to the characteristics "focused on one function" and "structural and behavioral autonomy".

To answer the **RQ3**, we simply compare the recall values obtained respectively from the experiment related to answering the **RQ1** and the **RQ2** (i.e., quality function with and without the evaluation of data autonomy).

The protocol for answering the **RQ4** is based on comparing the results generated by our identification approach using the software architect recommendations (i.e., semi-automatic) with those obtained without using them (i.e., fully automatic). The results of the manual identification remain of course the reference of confidence to calculate the distance between the two modes of identification (i.e., fully automatic or semi-automatic).

To answer the **RQ5**, we simply compare the identification results using the different software architect recommendations (e.g., exact number of microservices, interval of numbers, etc.). The recommendations that produce the most relevant microservices are the best ones to guide the identification process.

As explained earlier, to evaluate the produced microservices, we compare them with those identified manually. Thus, we classify the microservices obtained manually in three categories:

— **Category 1: Excellent microservices:** this category includes the microservices that exactly match the ones identified by our approach.
— **Category 2: Good microservices:** the microservices that can be obtained by at most three composition/decomposition operations of the ones identified by our approach are considered as good microservices. It is noteworthy that the composition and decomposition operations of microservices make it possible to adjust their granularity to that chosen by the software architect. Indeed, the granularity of a microservice is known to be strongly dependent on the style of each architect.
— **Category 3: Bad microservices:** they are the ones that are neither in the first nor the second categories.

5.1.2.2 Experimental Protocol for the Quantitative Evaluation of Microservices

We propose to evaluate our approach quantitatively based on the functional independence quality criterion presented in [76]. This criterion qualifies whether microservices can have their own functionalities independently. It has been proposed while taking into account the fact that a microservice should be functionally cohesive and decoupled from others [106].

The functional independence criterion relies on five metrics collected from existing works. They are CoHesion at Message level (*CHM*), CoHesion at Domain

level (*CHD*), InterFace Number (*IFN*), OPeration Number (*OPN*) and InteRaction Number (*IRN*). *CHM* and *CHD* evaluate the functional cohesion of microservices, whereas *IFN*, *OPN*, and *IRN* measure the coupling between them.

These metrics are evaluated relying mainly on microservice interfaces and their operations. The operations represent the methods implicated in the interactions between microservices. The interfaces correspond to the classes of the monolithic application containing these methods. It is noteworthy that to evaluate the functional independence of the identified microservices by our approach, the interfaces of each microservice as well as their operations were identified.

In [76], to present these metrics, the authors defined several parameters:

— $Microservices = \bigcup_{i=1,..,K} Microservice_i$
— $Microservice_i = (S\_C_i, S\_I_i)$
— $S\_C_i = \{C_{i1}, .., C_{ik_i}\}$ represents the classes of the $Microservice_i$.
— $S\_I_i = \{I_{i1}, .., I_{in_i}\}$ is the set of provided interfaces by $Microservice_i$. Each interface I_i has a set operations Op_i: $I_i = \{Op_{i1}, .., Op_{il_i}\}$.
— $Op_i = (res_i, name_i, pas_i)$ is an operation provided by an interface I_i. It has returned parameters $res_i = \{r_{i1}, .., r_{ip_i}\}$, a name $name_i$ and input parameters $pas_i = \{p_{i1}, .., p_{im_i}\}$.
— T_{Op_i} is the set of domain terms recovered from $name_i$ of Op_i. The recovery of these terms is based on the assumption that the names of the operations follow standard naming conventions. For instance, in Java [1], if a method name consists of one word, it should be a verb starting with a lowercase letter. Otherwise, the first letter of each internal word is capitalized.

Measuring CHM

CHM evaluates the average cohesion of microservice interfaces at message level. It represents a variant of Message-Level Cohesion LoC_{msg} [19]. Since microservices should be functionally cohesive, the higher CHM is the better. It is measured as follows:

$$CHM = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHM_j}{\sum_{i=1}^{K} n_i} \tag{5.1}$$

Where CHM_j computes the cohesion of a microservice interface I_j at message level as shown in Equation 5.2.

1. https://www.oracle.com/technetwork/java/codeconventions-135099.html

$$CHM_j = \begin{cases} \frac{\sum_{(k,m)} f_{simM}(Op_k, Op_m)}{|I_j|*(|I_j|-1)/2} & if \quad |I_j| \neq 1 \\ 1 & if \quad |I_j| = 1 \end{cases} \tag{5.2}$$

— n_i: number of provided interfaces by the $Microservice_i$.

— K: number of identified microservices.

— Op_k and Op_m are operations of I_j such that $k \neq m$.

— f_{simM} measures the similarity between two operations at message level. It is computed as the average similarity in term of input and returned parameters as follows:

$$f_{simM}(Op_k, Op_m) = \frac{(\frac{|res_k \bigcap res_m|}{|res_k \bigcup res_m|} + \frac{|pas_k \bigcap pas_m|}{|pas_k \bigcup pas_m|})}{2} \tag{5.3}$$

Measuring CHD

CHD evaluates the average cohesion of microservice interfaces at domain level. It represents a variant of Domain-Level Cohesion LoC_{dom} [19]. Similarly to *CHM*, the higher *CHD* is, the better. It is computed as follows:

$$CHD = \frac{\sum_{j=1}^{\sum_{i=1}^{K} n_i} CHD_j}{\sum_{i=1}^{K} n_i} \tag{5.4}$$

Where:

— CHD_j evaluates the cohesion of a microservice interface I_j at domain level using the following equation:

$$CHD_j = \begin{cases} \frac{\sum_{(k,m)} f_{simD}(Op_k, Op_m)}{|I_i|*(|I_i|-1)/2} & if \quad |I_i| \neq 1 \\ 1 & if \quad |I_i| = 1 \end{cases} \tag{5.5}$$

— f_{simD} measures the similarity between two operations at domain level. It is evaluated as follows:

$$f_{simD}(Op_k, Op_m) = \frac{|T_{Op_k} \bigcap T_{Op_m}|}{|T_{Op_k} \bigcup T_{Op_m}|} \tag{5.6}$$

Measuring IFN

IFN denotes the average number of interfaces provided by the identified microservices. It is inspired by [8]. Since microservices should be functionally decoupled, the lower IFN is, the better. It is evaluated as follows:

$$IFN = \frac{1}{K} \left| \bigcup_{i=1,..,K} S\_I_i \right| \tag{5.7}$$

Measuring OPN

OPN represents the number of operations provided by the identified microservices. The lower OPN is, the better. It is measured as follows:

$$OPN = \left| \bigcup_{i=1,..,\sum_{i=1}^{K} n_i} I_i \right| \tag{5.8}$$

Measuring IRN

IRN denotes the number of method calls between two microservices. It is evaluated as follows:

$$IRN = \sum_{Op_i, Op_j} w_{i,j} \tag{5.9}$$

Where $w_{i,j}$ represents the frequency of calls from Op_i to Op_j. Both operations are implicated in interactions between microservices. Similarly to IFN and OPN, since microservices should be functionally decoupled, the lower IRN is, the better.

Measuring Functional Independence to Answer Research Questions

The presented metrics above enable determining the functional independence of microservices based on their values. For instance, high values of *CHD* and *CHM* indicates that the microservices are highly cohesive. Whereas, low values of *IFN*, *IRN*, and *OPN* imply that they are highly decoupled from each other. As specified in [76], several microservice identification approaches can be compared by determining the number of metrics in which an approach outperforms the other ones.

To answer **RQ1** relying on functional independence measurement, we used

our plug-in to partition automatically (i.e., without architect recommendations) two Java applications. They will be presented in Section 5.1.3.2. Then, we evaluated the functional independence of the produced microservices. To better interpret results, we compare them with the ones obtained during the evaluation of the identified microservices from the same applications, using two well-known state-of-the-art approaches.

The protocol for answering the **RQ2** is similar to the one used to answer **RQ1** with only one difference: we set our plug-in to identify microservices based on the sub-function *FStructureBehavior* related only to the characteristics "focused on one function" and "structural and behavioral autonomy".

To answer the **RQ3**, we simply compare the results of functional independence measurement of the identified microservices with and without considering data autonomy.

The protocol for answering the **RQ4** is based on comparing the results of functional independence evaluation of the identified microservices by our approach with and without architect recommendations.

To answer the **RQ5**, we compare the results of functional independence measurement of the identified microservices relying on the different software architect recommendations.

5.1.3 Validating the Identification Based on a Clustering Algorithm

5.1.3.1 Qualitative Evaluation

In this section, we firstly outline the Java applications used to validate our approach qualitatively. After that, we present the microservice identification results from these applications using the clustering algorithm. Finally, we interpret the obtained results.

Data Collection

To have a codebase for partitioning OO applications into microservices, we collected several Java projects from *GitHub*. These projects have different sizes: small (*FindSportMates* [2]), average (*SpringBlog* [3]), and relatively large (*Inventory-*

2. https://github.com/chihweil5/FindSportMates
3. https://github.com/Raysmond/SpringBlog

ManagementSystem [4]). The source code of these applications used in our experiment, as well as their libraries, have been gathered in `https://seafile.lirmm.fr/d/2bb141de92c9420092b9/`. Table 5.1 provides some metrics on these applications.

Table 5.1 – *Applications metrics*

Application	No of classes and interfaces	No of classes representing database tables	Code size (LOC)
FindSportMates	17	2	895
SpringBlog	43	5	1617
InventoryManagementSystem	104	19	13449

FindSportMates is an application which allows users to find groups of people with whom they can play certain sports. Users can create a sport group or join existing ones. *SpringBlog* is a straightforward and clean-design blog system implemented with Spring Boot. It is powered by several frameworks and third-party projects (e.g., Spring MVC, Spring JPA, etc.). *InventoryManagementSystem* is an application that supports the main inventory management operations: getting items from different vendors, storing them in warehouses, selling or sending them to a third party, stocking ledger, etc.

Microservices Identification Results

The source code of each of the previous applications was partitioned into a set of clusters. Each cluster consists of one or more classes and corresponds to a microservice. Table 5.2 shows the number of microservices obtained relying respectively on the proposed fully automatic and semi-automatic clustering algorithms. Whereas, Table 5.3 presents the measurement results (i.e., minimum, maximum, and average values) of *FMicro*, *FStructureBehavior*, and *FData* of the identified microservices. It is noteworthy that to facilitate interpreting the measurement results, before printing them, the plug-in normalizes the values of *FStructureBehavior* and *FData* by dividing them on the maximum value, if it is higher than 1. After that, the plug-in computes *FMicro* based on the normalized values. The normalization does not impact the identification process.

For example, if we have the following measurements and coefficient weights:

— *FStructureBehavior(micro)*= 0.5

— *FData(micro)*= 0.9

4. `https://github.com/gtiwari333/java-inventory-management-system-swing-hibernate-nepal`

Table 5.2 – *Number of identified microservices from FindSportMates, SpringBlog, and InventoryManagementSystem applications*

Microservice identification			**Number of microservices**		
			FindSport Mates	Spring Blog	Inventory Management System
Automatic identification	Based on *FMicro*		3	7	16
	Based only on *FStructureBehavior*		2	5	14
Semi-automatic identification	Gravity centers	Entire set	3	10	20
		Sub-set	3	8	16
	Number of microservices	Exact number	3	10	20
		Interval of number	2	9	17
	Exact number and a sub-set of gravity centers		3	10	20

— *Max(FData)*= 1.2
— *Max(FStructureBehavior)*= 0.8
— α=1 and β= 1

Since the maximum value of *FData* measurement results is higher than 1, it is normalized. The printed values by the plug-in are the following:

— *NormalisedFData(micro)*=0.9/1.2= 0.75
— *FStructureBehavior(micro)*=0.5
— *NormalizedMax(FData)*= 1.2/1.2=1
— *Max(FStructureBehavior)*= 0.9
— *FMicro(micro)* = (α*FStructureBehavior(micro)+β*NormalisedFData(micro))/(α+β)
= (0.5+0.75)/2
FMicro(micro) = 0.625

We normalized the values to guarantee that the maximum is always 1. Nevertheless, we did not make sure that the minimum is always 0 since in most cases, the obtained minimum values are not lower than 0. To guarantee that the values are between 0 and 1, we can normalize them as follow:

$$normalizedValue(value) = \frac{value - mimimeum}{maximeum - minimeum} \tag{5.10}$$

The results of classifying the identified microservices based on our protocol (Section 5.1.2.1) are described in Table 5.4 and expressed in term of recall in Table 5.5. Recall assesses the ratio between the number of excellent and good microservices to the number of the manually identified ones.

Table 5.3 – *Measurement results of FMicro, FStructureBehavior, and FData*

Microservice identification					**Applications**		
					FindSport Mates	Spring Blog	Inventory Management System
Automatic identification	Based on *FMicro*		FMicro	Min	0,78	0	0
				Max	0,9	0,67	1
				Avg	0.83	0.25	0.16
			FStructure Behavior	Min	0.58	0	-0.73
				Max	1	0.51	1
				Avg	0.86	0.15	0.08
			FData	Min	0.71	0	0
				Max	1	1	1
				Avg	0.83	0.3	0.18
	Based only on *FStructureBehavior*		FMicro	Min	0.9	0	0
				Max	1	0.51	0.62
				Avg	0.95	0.13	0.2
			FStructure Behavior	Min	0.9	0	0
				Max	1	0.51	0.62
				Avg	0.95	0.13	0.2
			FData	Min	/	/	/
				Max	/	/	/
				Avg	/	/	/
Semi-automatic identification	Gravity centers	Entire set	FMicro	Min	0.1	0	0
				Max	0.92	0.5	0.52
				Avg	0.61	0.2	0.19
			FStructure Behavior	Min	0.38	0	0
				Max	1	1	1
				Avg	0.76	0.2	0.12
			FData	Min	0	0	0
				Max	0.9	0.72	0.73
				Avg	0.56	0.21	0.27
		Sub-set	FMicro	Min	0.1	0	0
				Max	0.63	0.5	0.67
				Avg	0.42	0.18	0.14
			FStructure Behavior	Min	0.39	0	0
				Max	1	1	1
				Avg	0.75	0.28	0.16
			FData	Min	0	0	0
				Max	0.5	0.66	1
				Avg	0.31	0.08	0.13
	Number of microservices	Exact number	FMicro	Min	0.78	0	0
				Max	0.9	0.52	0.55
				Avg	0.83	0.25	0.15
			FStructure Behavior	Min	0.58	0	0
				Max	1	1	1
				Avg	0.86	0.22	0.13
			FData	Min	0.71	0	0
				Max	1	0.72	0.73
				Avg	0.83	0.29	0.18
		Interval of number	FMicro	Min	0.63	0	0
				Max	0.87	0.51	0.54
				Avg	0.75	0.24	0.14
			FStructure Behavior	Min	0.97	0	0
				Max	1	1	1
				Avg	0.98	0.25	0.15
			FData	Min	0.51	0	0
				Max	0.82	0.72	0.73
				Avg	0.67	0.24	0.13
	Exact number and a sub-set of gravity centers		FMicro	Min	0.1	0	0
				Max	0.92	0.5	0.5
				Avg	0.61	0.13	0.21
			FStructure Behavior	Min	0.38	0	0
				Max	1	1	1
				Avg	0.76	0.2	0.15
			FData	Min	0	0	0
				Max	0.9	0.66	0.73
				Avg	0.56	0.07	0.27

Table 5.4 – *Microservice classification results*

Microservice identification				**Applications**		
				FindSport Mates	Spring Blog	Inventory Management System
Automatic identification	Based on FMicro		Number of excellent microservices	1	0	1
			Number of good microservices	2	8	15
			Number of bad microservices	0	2	4
	Based on FStructure Bahavior		Number of excellent microservices	0	0	0
			Number of good microservices	3	7	13
			Number of bad microservices	0	3	7
Semi-automatic identification	Gravity centers	Entire set	Number of excellent microservices	1	2	5
			Number of good microservices	2	8	15
			Number of bad microservices	0	0	0
		Sub-set	Number of excellent microservices	0	1	5
			Number of good microservices	3	8	14
			Number of bad microservices	0	1	1
	Number of microservices	Exact number	Number of excellent microservices	1	1	2
			Number of good microservices	2	7	15
			Number of bad microservices	0	2	3
		Interval of number	Number of excellent microservices	0	0	1
			Number of good microservices	3	8	16
			Number of bad microservices	0	2	3
	Exact number and a sub-set of gravity centers		Number of excellent microservices	1	3	5
			Number of good microservices	2	6	14
			Number of bad microservices	0	1	1

Table 5.5 – *Recall measurement*

Microservice identification			**Recall**		
			FindSport Mates	Spring Blog	Inventory Management System
Automatic identification	FMicro		100%	80%	80%
	FStructureBehavior		100%	70%	65%
Semi-automatic identification	Gravity centers	Entire set	100%	100%	100%
		Sub-set	100%	90%	95%
	Number of microservices	Exact number	100%	80%	85%
		Interval of number	100%	80%	85%
	Exact number and a sub-set of gravity centers		100%	90%	95%

To better understand and since *FindSportMates* is a small application, we will consider it as an example. Figure 5.1 represents its class diagram. It was recovered from the source code of this application using the tool *ObjectAid UML Explorer* [5]. For clarity and briefness, the operations are not represented in the diagram. Figure 5.2 displays the identified microservices by our approach (i.e., fully automatic identification), as well as their data manipulations. Whereas, Figure 5.3 shows the manually identified microservices.

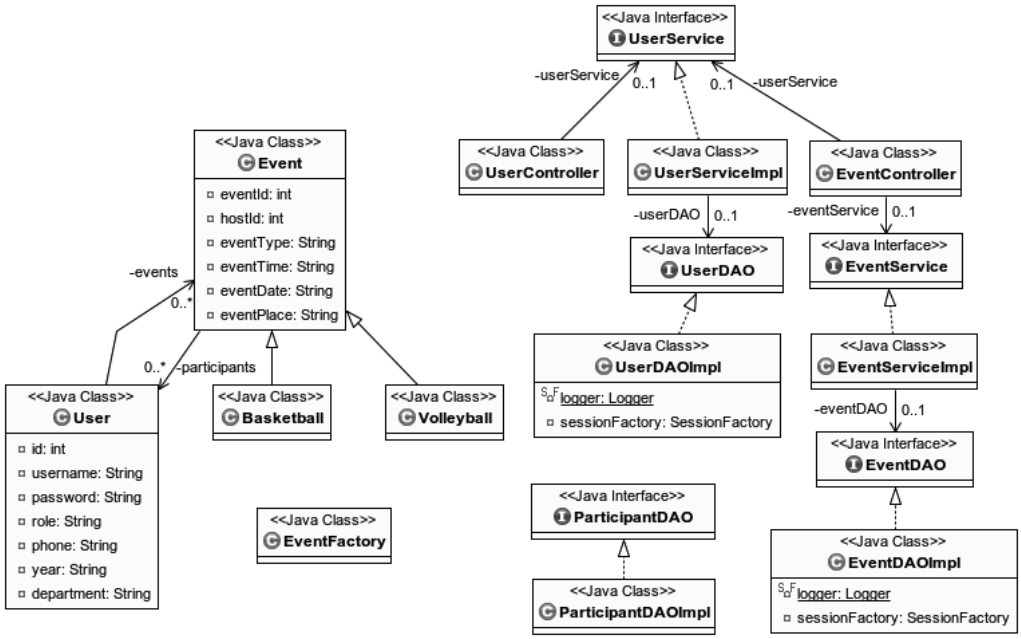

Figure 5.1 – *Class diagram of FindSportMates application*

5. https://www.objectaid.com/home

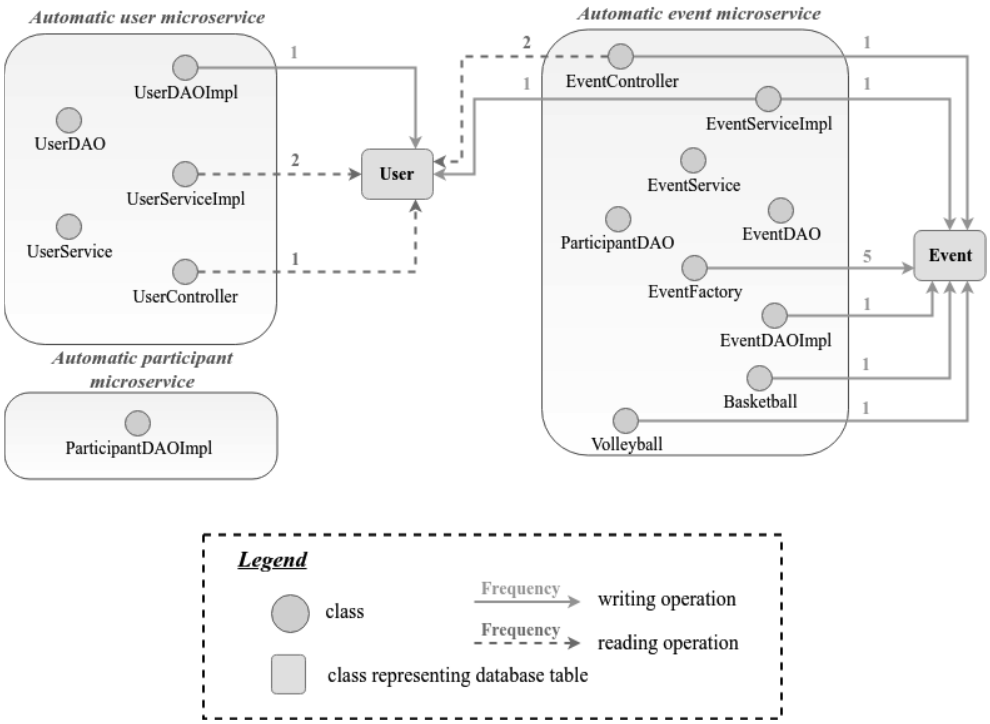

Figure 5.2 – *Microservice identification results from FindSportMates application*

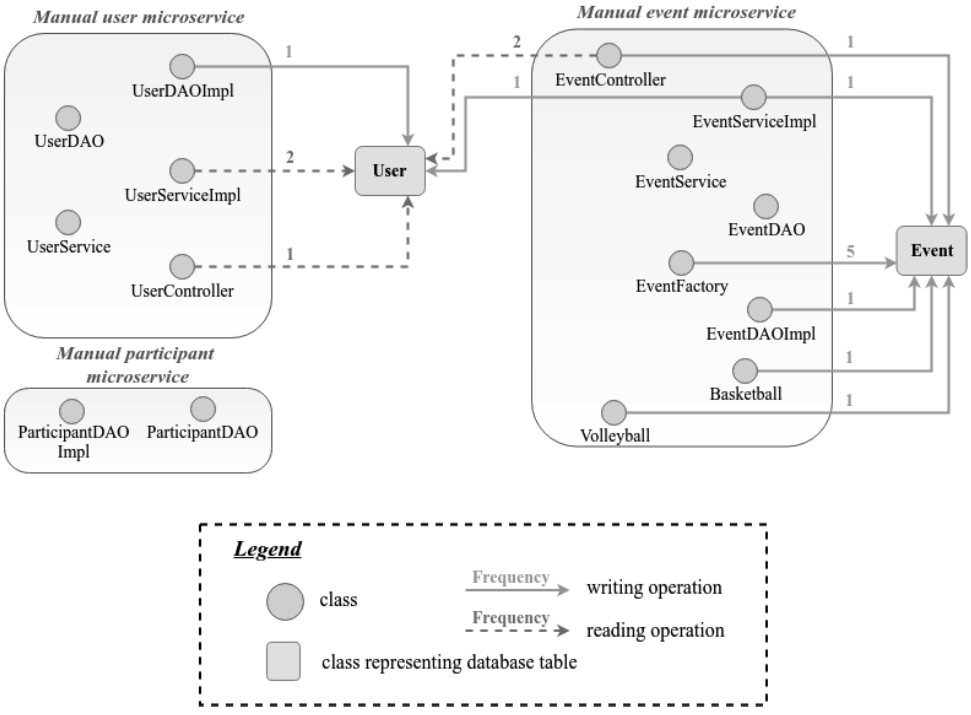

Figure 5.3 – *Manually identified microservices from FindSportMates application*

Our approach identified three microservice: *Automatic user microservice*, *Automatic participant microservice*, and *Automatic event microservice*. *Automatic user microservice* is identical to the one identified manually. As shown in Figure 5.4, the decomposition of *Automatic event microservice* produces *Manual event microservice* (i.e., one operation), and *Automatic event sub-microservice*. The later can be composed with *Automatic participant microservice* to produce the *Manual participant microservice* (i.e., two operations). Therefore, our approach identified two good microservices and an excellent one.

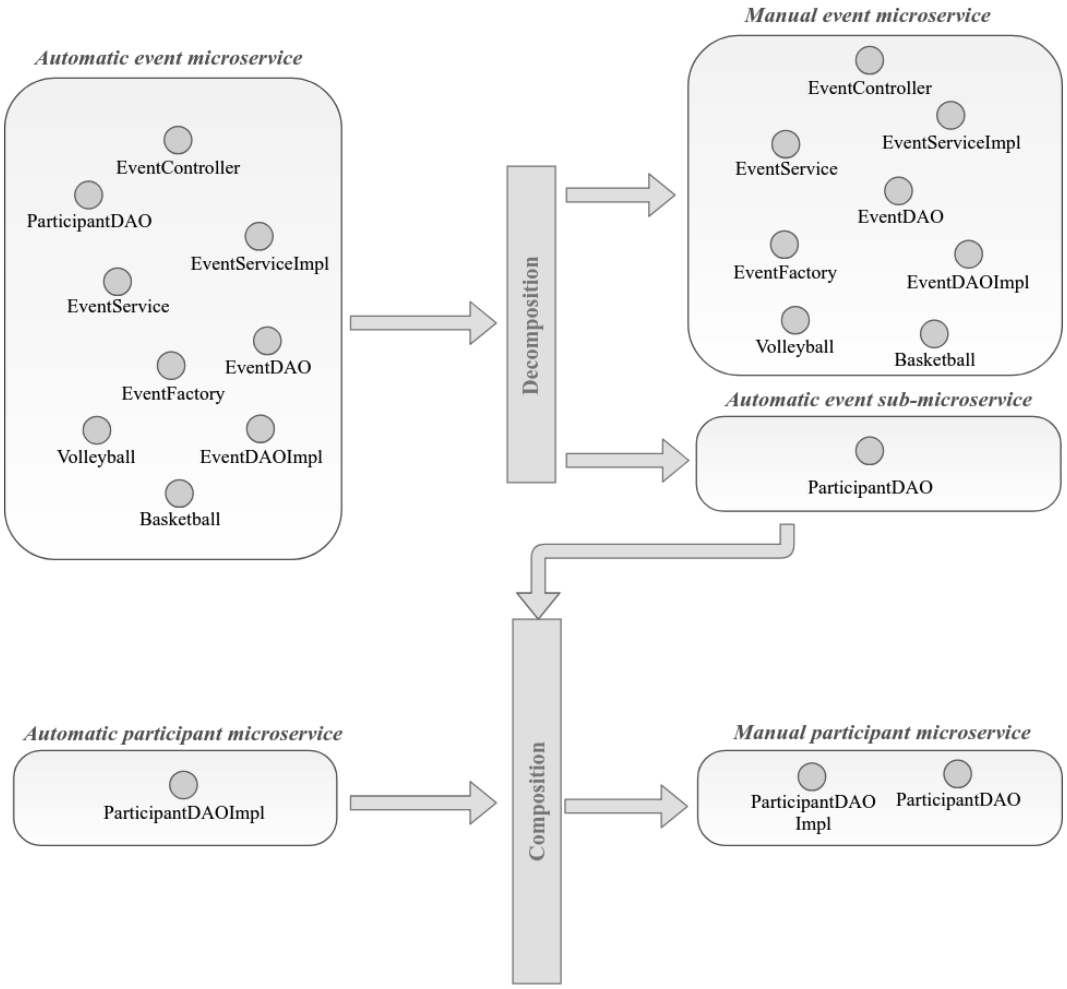

Figure 5.4 – *Composition and decomposition of the automatically generated microservices to obtain the manually identified ones*

We analyzed the source code of *FindSportMates application* to understand why only one excellent microservice was identified. We found out that *ParticipantDAO* and *ParticipantDAOImpl* are not used by the other classes of the application. In

other words, they can be deleted without any impact. Moreover, the body of the methods of *ParticipantDAOImpl* is empty.

Interpreting Results

Now that the microservice identification results have been presented, we will interpret them. Firstly, we will discuss the obtained number of microservices by the different variants of our approach. After that, we will interpret the measurement results of *FStructureBehavior*, *FData*, and *FMicro*. Finally, we will discuss the classification results.

(a) Number of identified microservices: on the one hand, the number of produced microservices from the three applications *FindSportMates*, *SpringBlog*, and *InventoryManagementSystem* based on *FMicro* (resp., 3, 7, and 16) is higher than the one obtained without evaluating data autonomy (resp., 2, 5, and 14). Moreover, it is closer to the exact number that we specified (resp., 3, 10, and 20). On the other hand, the number of generated microservices relying on software architect recommendations is equal to or higher than the one obtained automatically, except for *FindSportMates*. The identification based on an interval of numbers produced two microservices. We specified the interval [2, 4]. Since choosing a number within this interval is random, it was lower than the automatically obtained one. Still, this number is close to the exact one.

These results show that generally using more information (i.e., extracted from source code or provided by software architect) increases the chances of identifying the exact number of microservices. Note that here, we focus on the recommendations that do not precise the exact number (i.e., sub-set of gravity centers and interval of numbers), since the other ones produce the desired number of microservices.

(b) Measurement results of FStructureBehavior, FData, and FMicro: firstly, the values of *FStructureBehavior* depend mainly on the values of the metrics (i.e., internal coupling, internal cohesion, and external coupling) used in its sub-functions (i.e., *FOne* and *FAutonomy*). The values of internal and external coupling are affected by the size of applications. For this reason, the average values of *FStructureBehavior* of the identified microservices from *FindSportMates* application (between 0.75 and 0.98) are higher than those obtained from *SpringBlog* (between 0.13 and 0.28), and *InventoryManagementSystem* (between 0.08 and 0.2). This is due to the fact that large applications contain a higher number of total calls (i.e., *TotalNbCalls* used to measure *CouplingPair* as explained in Section 3.3.1.2). Therefore, even if the classes of a microservice contain methods calling each other frequently, the number of calls is small compared to the total. Thus, the values of internal coupling will not be high. Nevertheless, they will definitely be higher than those obtained from microservices containing independent classes (i.e., do

not call each other's methods). Furthermore, since we do not consider attribute accesses by the methods of the same class (Section 3.3.1.2), the cohesion values are not high.

Note that during the identification process, we used double-precision floating-point values (e.g., 0.000724637681159420, etc.). For clarity and briefness, we printed only two digits after the decimal point. Therefore, the minimum value of *FStructureBahavior* is generally 0. This value is produced when the number of internal structural dependencies between the classes of a microservice is almost equal to the number of external ones (i.e., the value of *FOne* is very closed to the value of *FAuotonomy*). There is only one minimum value lower than 0 (- 0.73). This value was obtained because the standard deviation between the coupling values of the corresponding microservices classes is high. The following question may be asked: why this microservices was identified even though the measurement result of *FStructureBehavior* is low? The answer is: it was identified because its classes manipulate the same data, and we gave a higher coefficient weight to *FData*.

Secondly, the values of *FData* depend mainly on the number of the classes of the OO application manipulating data. They also depend on the frequency of manipulation. For instance, most classes of *FindSportMates* application read/write data (Figure 5.2). Therefore, for this application, the average values of *FData* obtained relying on the different variants of our approach are between 0.31 and 0.83. The low minimum values of *FData* (i.e., Min= 0) are related to the microservices that do not manipulate any data (i.e., their classes do not read/write data). The high number of these microservice decreases the average value of *FData*. This is the case with *SpringBlog* and *InventoryManegementSystem*.

Finally, the values of *FMicro* depend primarily on the ones obtained by evaluating its sub-functions (i.e., *FStructureBehavior* and *FData*). When their average values are high, the average of *FMicro* is also high and vice versa. Similarly, high maximum (resp., minimum) values of *FStructureBehavior* and *FData* produce high (resp., low) maximum (resp., minimum) values of *FMicro*. For instance, since the averages of the *FStructureBehavior* and *FData* obtained from *InventoryManagementSystem* based on the exact number of microservices are low (resp., 0.13 and 0.18), the average of *FMicro* is also low (0.15).

(c) Classification of the identified microservices: firstly, the recall values related to the results obtained based only on the quality function *FMicro* (i.e., without architect recommendations) are equals to or greater than 80% (80% and 100%). This shows that a large part of the produced microservices are those identified manually. However, for *InventoryManagementSystem*, the number of bad microservices is relatively high (i.e., 4 bad microservices). When analyzing results, we found out that usually, the bad microservices are the ones containing

utility classes. Manually, we identified them as microservices because they participate in the realization of several functionalities of *InventoryManagementSystem* application. The utility classes generally do not use each other's methods or attributes. Moreover, they do not manipulate any data. Therefore, our approach could not identify them as microservices.

Secondly, the recall values obtained relying on the sub-function *FStructureBehavior* are between 65% and 100% (65%, 70%, and 100%). They are equal to or less than those obtained based on the entire quality function *FMicro*. Nevertheless, they remain high. An analysis of Table 5.1 allows us to understand that the same values are related to the application *FindSpotMates* that does not have many persistent data.

Thirdly, the recall values obtained using software architect recommendations are higher than 80%. These values are equal to or higher than those obtained without using software architect recommendations. Furthermore, for *SpringBlog* and *InventoryManagementSystem*, the highest values are produced relying on software architect recommendations (100%). For *FindSportMates*, the highest value is 100%. Nevertheless, since this application is small, it was partitioned correctly with and without software architect recommendations. Additionally, by analyzing the results of Table 5.5, we can see that precise (i.e., exact number) and complete (i.e., entire set of gravity centers) produce better results (i.e., higher recall values or more excellent microservices). For instance, the recall values related to the results obtained from *SpringBlog* relying on the exact number and interval of numbers are the same (80%). However, the identification guided by the exact number produced 1 excellent microservices and 7 good ones, whereas the one guided by an interval of numbers produced 8 good microservices.

Finally, the recall values obtained based on more recommendations (i.e., entire-set of gravity centers, which imply the availability of the number of microservices, or the exact number and a sub-set of gravity centers) are equal or higher than those obtained based on one recommendation (i.e., sub-set of gravity center, exact number or interval of numbers). Usually, when the values are the same, more recommendations produce a higher number of excellent microservices. For instance, the identification results related to *SpringBlog* application based on software architect recommendations can be ordered as follows: 1) entire set of gravity centers (100%), 2) exact number and a sub-set of gravity centers (90% and 3 excellent microservices), 3) sub-set of gravity centers (90% and 1 excellent microservices), 4) exact number (80% and 1 excellent microservice) and 5) interval of numbers (80% and no excellent microservice). It is noteworthy that the identification guided by the entire set of gravity centers produce the best results for the three applications (100%).

5.1.3.2 Quantitative Evaluation and Comparison with Some Existing Approaches

Our evaluation presented in the previous section is qualitative (i.e., associate a qualification to the identified microservices). In this section, we assess our approach quantitatively relying on the functional independence quality criterion. As explained earlier, this criterion is based on five metrics. To better interpret the values of these metrics obtained from the assessment of the identified microservices by our approach, we compare them with those collected during the evaluation of two state-of-the-art approaches among the well-known ones. They are Microservice Extraction Model (*MEM*) [100] and Functionally-oriented Microservice Extraction (*FoME*) [76]. We selected these approaches for the following main reasons:

— Both approaches have been recently proposed.

— Similarly to our approach, they aim to identify microservices as clusters of classes.

— To validate *FoME*, Jin et al. [76] have proposed the functional independence quality criterion. They relied on it to compare the identified microservices by their approach with other ones, including *MEM* [100]. More precisely, in [100], the authors have proposed three coupling strategies (i.e., logical, semantic, and contributor) and embed them in a graph-based clustering algorithm. Only the semantic coupling strategy was compared in [76] since the other ones cover a very limited number of the classes of the decomposed monolithic applications. The measurement results have been presented in the corresponding paper. Therefore, we simply have to identify microservices using our approach from the same applications utilized by Jin et al., implement as well as measure functional independence and then compare results.

In the remainder of this section, we firstly introduce the used applications. After that, we present the evaluation results, and compare them.

Data Collection

Among the four applications used in [76], two were collected to be an experimentation base for our approach: *JPetStore* and *SpringBlog*. *JPetStore*[6] is a web application for an e-commerce pet store. *SpringBlog*[7] is a blogging system. It can be considered as another version of the application presented in Section 5.1.3.1. On GitHub, it is specified that this *SpringBlog* application was forked from the one presented earlier. Table 5.6 provides some metrics on these applications.

6. https://github.com/wj86/jpetstore-6
7. https://github.com/wj86/SpringBlog

Table 5.6 – *Applications metrics*

Application	No of classes and interfaces	No of classes representing database tables	Code size (LOC)
JPetStore	24	9	1441
SpringBlog	93	10	3802

Results of Functional Independence Evaluation

The source code of each of the previous applications was partitioned into a set of microservices. Each one of them consists of one or more classes. To evaluate the identified microservices quantitatively and compare our approach with *MEM* and *FoME*, we measured their functional independence. Table 5.7 displays the measurement results. As explained earlier, the higher (resp., lower) *CHD* and *CHM* (resp., *IFN*, *OPN*, and *IRN*) are, the better.

Comparison and Results Interpretation

Table 5.7 – *Functional independence measurement results*

Appli-cation	Metrics	Approach								
		MEM	FOME	Our approach						
				Fully automatic		Semi-automatic				
						Gravity centers		Number of microservices		Exact number + sub-set of gravity centers
				FMicro	FStructure Behavior	Entire set	Sub-set	Exact number	Interval of number	
JPet Store	CHM	0.5-0.6	0.7-0.8	0.3-0.4	0.4-0.5	0.7-0.8	0.3-0.4	0.3-0.4	0.3-0.4	0.3-0.4
	CHD	0.6-0.7	0.6-0.7	0.6-0.7	0.6-0.7	0.7-0.8	0.6-0.7	0.6-0.7	0.6-0.7	0.6-0.7
	IFN	0.6667	1	1.6667	3	2	2.5	1.6667	1.6667	3
	OPN	39	22	11	22	14	22	11	11	18
	IRN	48	35	14	24	15	31	14	14	27
	Total	1	1	2	0	2	0	2	2	0
	Total*			3	3	4	3	3	3	3
	No micro	3	3	3	2	3	4	3	3	3
Spring Blog	CHM	0.7-0.8	0.7-0.8	0.8-0.9	0.3-0.4	0.9-1	0.9-1	0.8-0.9	0.7-0.8	0.8-0.9
	CHD	0.8-0.9	0.8-0.9	0.5-0.6	0.1-0.2	0.8-0.9	0.7-0.8	0.6-0.7	0.4-0.5	0.8-0.9
	IFN	1.4286	1.0000	1.3333	2	1.5714	1.6667	0.5714	0.5000	1.2857
	OPN	21	7	6	15	14	14	9	8	12
	IRN	30	26	12	16	26	26	12	10	23
	Total	1	1	1	0	2	1	0	1	1
	Total*			3	1	3	2	3	3	3
	No micro	7	7	3	2	7	6	7	6	7

— Total: the number of metrics, in which an approach outperforms the others. Here, we compare even the variants of our approach.

— Total*: the number of metrics, in which a variant of our approach outperforms MEM and FOME.

— No micro: number of identified microservices.

— Highlighted in red and underlined: best results.

Firstly, based on the *Total** row, our automatic approach outperforms both *MEM* and *FoME*. More precisely, the identified microservice by our approach from *JPetStore* have the same value of *CHD* and lower values of *OPN* as well as *IRN*, compared to those obtained by *MEM* and *FoMe*. For *SpringBlog*, our approach produced lower *OPN* as well as *IRN* and higher *CHM* values.

Secondly, the produced microservices relying on *FMicro* are better than those obtained based on the sub-function *FStructureBehavior*. However, for *JPetStore*, they still better than those produced by *MEM* and *FoME* (i.e., the same values of *CHD* and *OPN* and lower value of *IRN*). For *SpringBlog*, only a lower *IRN* was obtained. This shows the importance of data autonomy evaluation.

Thirdly, based on the *Total** row, generally the use of software architect recommendations either produce the same total obtained by the automatic approach (i.e., 3 for *JPetStore* and 3 for *SpringBlog*) or increase it (i.e., 4 for *JPetStore*), except for the obtained total based on a sub-set of gravity center from *JPetStore*. This is potentially because we miss-specified this sub-set.

Finally, the *Total** values obtained based on the different recommendations are almost always the same for each application, except in two cases: 1) for *JPetStore*, the entire set of gravity centers produced a better result and 2) for *SpringBlog*, the *Total** value obtained based on sub-set of gravity centers is lower. The entire set of gravity center is a complete and precise recommendation. Thus, using it enabled producing better results. The low value of *Total** obtained based on a sub-set of gravity centers is potentially because we miss-specified this sub-set.

5.1.4 Validating the Identification Based on a Genetic Algorithm

Our microservice identification approach enables identifying microservices based on a clustering algorithm or a genetic one. In the previous section, we showed the relevance of the obtained microservices based on the clustering algorithm. To validate the ones produced relying on the genetic one, our idea is to compare the results of the automatic identification based on both algorithms, since we already demonstrated the importance of taking into account architect recommendations. For that purpose, we identified microservices from the three applications presented in Table 5.1. The remainder of this section presents the obtained results as well as their interpretations.

5.1.4.1 Microservices Identification Results

The source code of each of the previous applications was partitioned into a set of clusters. Each cluster consists of one or more classes and corresponds to a microservice. Table 5.8 presents the measurement results (i.e., minimum, maximum, and average values) of *FMicro*, *FStructureBehavior*, and *FData* of the identified microservices.

To facilitate interpreting the measurement results, before printing them, the values of *FStructureBehavior* and *FData* are normalized by dividing them on the maximum value, if it is higher than 1. After that, *FMicro* is computed based on the normalized values. Note that all the results related to the clustering algorithm are the ones presented earlier in this section. We repeated them here to facilitate the comparison of the two algorithms.

Table 5.8 – *Measurement results of FMicro, FStructureBehavior, and FData*

Microservice identification			**Applications**		
			FindSport Mates	SpringBlog	InventoryManagement System
Clustering algoritm	FMicro	Min	0,78	0	0
		Max	0.9	0.67	1
		Avg	0.83	0.25	0.16
	FStructure Behavior	Min	0.58	0	-0.73
		Max	1	0.51	1
		Avg	0.86	0.15	0.08
	FData	Min	0.71	0	0
		Max	1	1	1
		Avg	0.83	0.3	0.18
Genetic algorithm	FMicro	Min	0.35	0.15	0.12
		Max	0.96	0.84	0.86
		Avg	0.57	0.41	0.53
	FStructure Behavior	Min	0.75	0.37	0
		Max	1	1	1
		Avg	0.9	0.65	0.41
	FData	Min	0.25	0	0
		Max	1	1	1
		Avg	0.52	0.33	0.55

The results of classifying the identified microservices based on our protocol (Section 5.1.2.1) are described in Table 5.9 and expressed in term of recall in Table 5.10. Recall assesses the ratio between the number of excellent and good microservices to the number of the manually identified ones.

Table 5.9 – *Microservice classification results*

Microservice identification		**Applications**		
		FindSport Mates	Spring Blog	Inventory Management System
Clustering algorithm	Number of excellent microservices	1	0	1
	Number of good microservices	2	8	15
	Number of bad microservices	0	2	4
Genetic algorithm	Number of excellent microservices	0	0	0
	Number of good microservices	3	9	19
	Number of bad microservices	0	1	1

Table 5.10 – *Recall measurement*

Microservice identification	**Applications**		
	FindSportMates	SpringBlog	InventoryManagementSystem
Clustering algorithm	100%	80%	80%
Genetic algorithm	100%	90%	95%

5.1.4.2 Interpreting Results

Measurement Results of FStructureBehavior, FData, and FMicro

The interpretations of the obtained measurements results of *FStructureBehavior*, *FData*, and *FMicro* (e.g., low values, high values, etc.) were presented in details in Section 5.1.3.1. Thus, we will not reinterpret them here. Still, it is noteworthy that overall, the genetic algorithm produced higher minimum, maximum, and average values of *FStructureBehavior*, *FData*, and *FMicro*.

Classification of the Identified Microservices

The recall values related to the results obtained based on the genetic algorithm are equals to or greater than 90% (90%, 95%, and 100%). This shows that a large part of the produced microservices are those identified manually. These values are equal to or higher than those produced by the clustering algorithm (80% and 100%). Furthermore, for *SpringBlog* and *InventoryManagementSystem*, the highest values are produced relying on the genetic algorithm (90% and 95%). For *FindSportMates*, the highest value is 100%. Nevertheless, since this application

is small, it was partitioned correctly by both algorithms. Based on the obtained recall results, the genetic algorithm outperforms the clustering one.

It is noteworthy that the genetic algorithm identified a lower number of bad microservices. However, it did not produce any excellent ones. This is likely because the population was randomly initialized. According to Chardigny [37], the better the initial population is, the better the produced results will be. Hence, initializing the population, for instance, relying on the clustering algorithm can profoundly improve results.

5.1.5 Answering Research Questions

To validate our microservice identification approach, this section aims to answer the research questions based on the obtained results. For that purpose, we will focus on those generated by the clustering algorithm. As demonstrated earlier, the genetic one produces better results. Therefore, if the obtained ones by the clustering algorithm allow validating our approach empirically, the generated results by the genetic algorithm will potentially enable a better validation.

5.1.5.1 Answers Based on the Qualitative Evaluation

Based on the qualitative evaluation of the obtained results by our approach, the research questions can be answered as follows:

1. Since a large part of the identified microservices without using the architect recommendations are those identified manually, we answer the **RQ1** as follows: the proposed quality function produces an adequate decomposition of an OO application into microservices.
2. Even though the obtained recall values based on the sub-function *FStructureBehavior* are equal to or lower than those produced by *FMicro*, they still high. Therefore, we answer the **RQ2** as follows: the definition of the quality function, without considering data autonomy, is adequate.
3. The obtained recall values based on *FMicro* are equal to or higher than those obtained relying on *FStructureBehavior*. Moreover, the same values are related to the application that does not have many persistent data. Hence, we answer **RQ3** as follows: the evaluation of data autonomy characteristic enhance the quality of microservices.
4. The recall values obtained using software architect recommendations are equal to or higher than those obtained without using them. Furthermore,

the best results are always produced relying on these recommendations. Therefore, the answer to **RQ4** is the following: the use of software architect recommendations enhance the identification results.

5. Since the best identification results are produced based on the entire set of gravity centers, we answer **RQ5** as follows: the software architect recommendations that generate the best decomposition of an OO application into microservices is the entire set of gravity centers.

5.1.5.2 Answers Based on the Quantitative Evaluation

Based on the quantitative evaluation of the obtained results by our approach, the research questions can be answered as follows:

1. Since our automatic identification, based on *FMicro*, outperforms both *MEM* and *FoME*, we answer the **RQ1** as follows: the proposed quality function produces an adequate decomposition of an OO application into microservices.
2. For *JPetStore*, the obtained results relying only on the sub-function *FStructureBehavior* are better than the ones produced by *MEM* and *FoME*. However, for *SpringBlog*, the existing approaches generate better values. Relying on these results, we cannot answer **RQ2** quantitatively unless we experiment on other applications.
3. The obtained results based on the quality function *FMicro* are better than the ones produced relying only on the sub-function *FStructureBahavior*. Therefore, we answer the **RQ3** as follows: the evaluation of data autonomy characteristic enhance the quality of microservices.
4. Generally, the use of software architect recommendations either produce the same results obtained by the automatic approach or improve them. Thus, the answer to **RQ4** is the following: the use of software architect recommendations can enhance the identification results.
5. Since, for both applications, the best evaluation results are produced based on the entire set of gravity centers, we answer **RQ5** as follows: the software architect recommendations that generate the best decomposition of an OO application into microservices is the entire set of gravity centers.

5.1.6 Threats to Validity

Our microservice identification approach is concerned by two types of threats to validity: internal and external. We will present the main ones in this section.

5.1.6.1 Threats to Internal Validity

The proposed approach may be affected by the following internal threats:

— Our decomposition of an OO application into microservices realizes a partition of the classes. Therefore, each class belongs to one and only one cluster, so it is part of the implementation of one and only one microservice. This may not reflect the reality of some applications where some classes may participate in the realization of several functionalities, and thus could be part of the implementation of several microservices. In this case, the results of the identification could be negatively impacted. Nevertheless, this threat is limited because it generally concerns only certain classes that the architect can duplicate to correct the concerned microservices.

— We rely on a hierarchical clustering algorithm to partition the classes of an OO application. This algorithm does not allow to obtain optimal values of the quality function. Indeed, some grouping choices may not be the best considering the whole process of grouping and not just the one at a given moment. Nevertheless, the algorithm makes it possible to obtain values close to optimal ones because this algorithm performs optimization at each clustering step.

5.1.6.2 Threats to External Validity

Our approach could be concerned with the following external threats:

— The quality of the OO source code can impact the results of the microservice identification. Indeed, besides the use of software architect recommendations, our approach analyzes, on the one hand, the relations between the entities of the code, and on the other hand, relationships between these entities and their manipulated data. Thus, for example, the microservices obtained from a source code that does not respect the basic rules of modularity will be of the same quality of the original source code. To reduce the impact of this factor, and obtain more relevant microservices, a possible refactoring of the monolithic applications enabling the improvement of their modularity may be necessary.

— The matching between the microservices that can be obtained by our approach and those obtained manually can vary according to the granularity of microservices obtained by a manual identification. Granularity is known to be dependent on the architectural style of the software architect. Depending on his style, an architect can manually adjust the granularity of

microservices obtained by our approach. This adjustment can be achieved by a limited number of microservices composition/decomposition operations.

— Our approach was experimented only on Java applications. To apply it on the ones implemented using other languages (e.g., C++, C#, etc.), we have to adapt it to take into consideration the specificities of these languages, such as multiple inheritance in C++.

— To compare our microservice identification against *FoME* and *MEM*, we evaluated the microservices identified by our approach and recovered the functional independence measurement provided in the paper for *FoME* and *MEM*. In other words, the three approaches were not evaluated using the same implementation. However, we implemented the functional independence measurement relying on the explanations provided in the paper [76] proposing it.

5.2 Experimentation and Validation of our Extraction Approach of Workflows from OO Applications

This section aims to present the conducted experiments to validate our workflow extraction approach. It starts with introducing the used applications, followed by the experimental protocol. After that, the workflow extraction results are presented and discussed. Finally, we analyze the internal and external threats.

5.2.1 Data Collection

To validate our workflow extraction approach, we used three applications: *eLib*, *PostOffice*, and *FindSportMates*. *eLib* is a Java application supporting the main functionalities operated in a library: 1) inserting and removing users/documents, 2) searching for users/documents, and 3) loan management. The code of this application is provided in [133]. *PostOffice* is also a Java application that determines for each postal item (i.e., letters, parcels, and express parcels) its postage fee and its reimbursement rate. Moreover, it allows printing the information of any chosen item. *FindSportMates* has been presented earlier in Section 5.1.3.1. Table 5.11 provides some metrics on these applications before applying extract method refactoring. It is noteworthy that to identify microservices, we used the refactored *FindSportMates* application since it is more structured.

Table 5.11 – *Applications metrics before applying extract method refactoring*

Application	No of classes	No of methods	Code size (LOC)
PostOffice	6	50	270
eLib	9	73	534
FindSportMates	12	92	796

5.2.2 Experimental Protocol

The validation of our workflow extraction approach is based on a prototype plug-in developed in Java. It carries out the extraction process defined in our approach. To demonstrate that the recovered workflow reflects the behavior of the analyzed OO application (i.e., correct workflow), we use two methods:

Running the Extracted Workflow

The idea is to execute both the extracted workflow and the corresponding OO application using the same test suite and compare results. If given the same inputs, the workflow and the OO application produce the same outputs, then the extracted workflow is correct. To be able to run the recovered workflow, we have proposed the implementation model shown in Figure 5.5. In this model, each task is an instance of the class *Task*. It has two data lists: a list of inputs and a list of outputs. The execution of any task requires invoking the method *run* on the corresponding instance. However, the method *run* of a primitive task is not the same as a composite one:

— ***Primitive task:*** the method *run* of a primitive task initializes inputs and invokes the corresponding method to this task.

— ***Composite task:*** in the method *run* of a composite task, instances corresponding to the enclosed elements in this task are created and their methods *run* are executed. Note that these elements are specified in the attribute *taskSubelements* of the class *CompositeTask*. Moreover, they can either be control constructs or other tasks. These elements are executed based on their order in the list *taskSubelements*.

Similarly, to run a control construct (i.e., if construct or while construct), an instance of the corresponding class to this construct (i.e., IfConstruct class or WhileConstruct class) is created, and the method *run* is invoked on this instance. The *run* method initializes control construct inputs and evaluates its condition. For example, in the case of while construct, if the condition is true, then instances

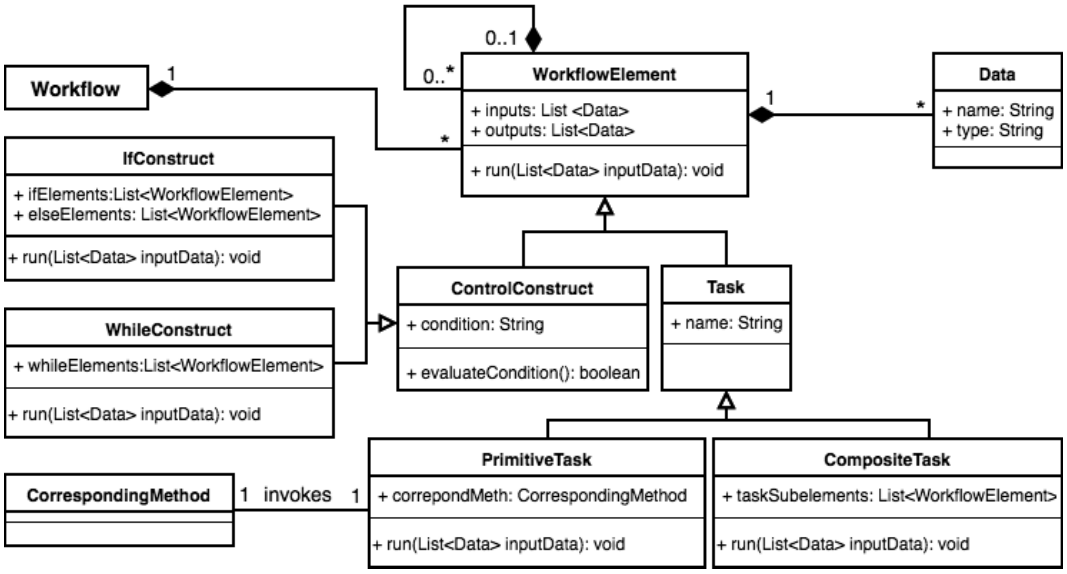

Figure 5.5 – *Workflow implementation model*

corresponding to the elements of the list *whileElements* are created, and their *run* methods are executed. In the case of if construct, when the condition is true (i.e., resp false), instances corresponding to the elements of the list *IfElements* (i.e., resp elseElements) are created, and their *run* methods are executed.

It is noteworthy that the generated implementation of the recovered workflow preserves the object entities of the original source code. In other words, object instances are not be transformed into primitive elements (i.e., the values of their attributes). Hence, the produced implementation is based on "task" entities connected by input and output data and whose control flow is explicitly specified at the architectural level. This means that the produced source code consists of task entities, implemented relying on the object entities of the original source code.

Manual Evaluation

The manual evaluation consists to extract the corresponding workflow to each of the analyzed applications manually and automatically (i.e., using our plug-in), and compare them.

5.2.3 Workflow Extraction Results and their Interpretations

We used our plug-in to generate the corresponding workflow to each of the analyzed OO applications. The first step in our process (i.e., task identification)

requires applying extract method refactoring. Table 5.12 provides some metrics on the three applications after applying this refactoring.

Table 5.12 – *Applications metrics after applying extract method refactoring*

Application	No of classes	No of methods	Code size (LOC)	% of the added methods
PostOffice	6	67	341	34%
eLib	9	113	788	54.79%
FindSportMates	12	121	895	31.52%

As we can notice, the number of methods after applying extract method refactoring increased by an average of *40.10%* with a standard deviation of *10.43*. This can be explained by the fact that new methods were created depending on the number of fragments to be extracted (i.e., fragments consisting of statements delimited by user method invocations that belong to the same block), in the source code of the OO applications.

Once the refactoring is done, the plug-in analyzes the source code to identify tasks. Table 5.13 shows the results in term of the number of primitive and composite tasks for each application, as well as the total number of identified tasks.

Table 5.13 – *Workflow extraction results*

Application	No of primitive tasks	No of composite tasks	Total
PostOffice	51	16	67
eLib	77	36	113
FindSportMates	91	30	121

To better understand, we will consider *eLib* as an example. Figure 5.6 represents its class diagram. It was recovered from the source code of this application using the tool *ObjectAid UML Explorer*. For clarity and briefness, the operations of each class were not represented in the diagram. Figure 5.7 displays the extracted workflow. It consists of seven tasks: six primitive ones and a composite task, which corresponds to the method *dispatchCommand*.

For each application, we executed both the code corresponding to the workflow and the one corresponding to the OO application relying on the same test suite (i.e., test cases). Both executions produced the same results. Note that the test suites for the analyzed applications were specified manually while taking

into account their functionalities. As an example, Table 5.14 presents the test cases for *eLib* application, which offers 15 fine-grained functionalities.

Moreover, we identified workflows manually from each application and compared them with the ones generated by our plug-in. Similarly, we found out that they are the same.

Based on the obtained results from the conducted experiments, the extracted workflows from OO source code reflects the behavior of the analyzed OO application (i.e., correct workflow).

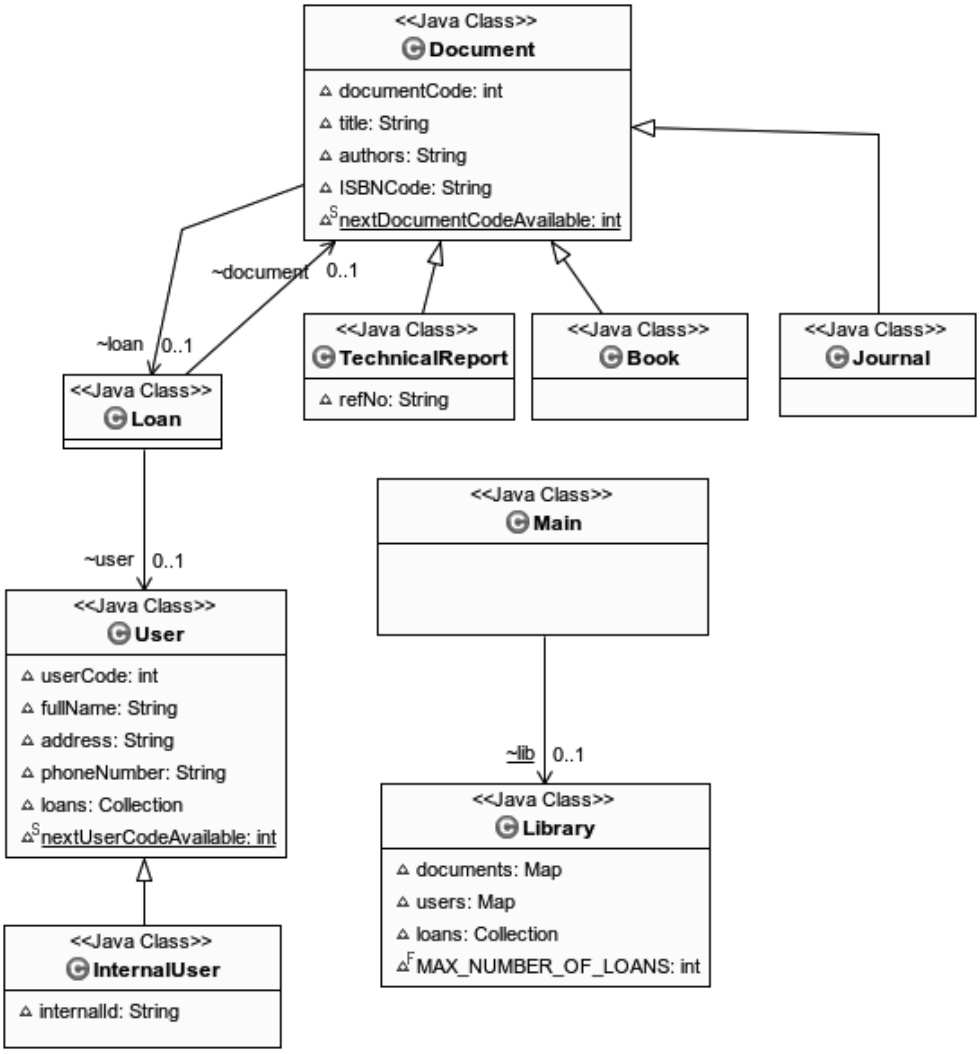

Figure 5.6 – *Class diagram of the eLib application*

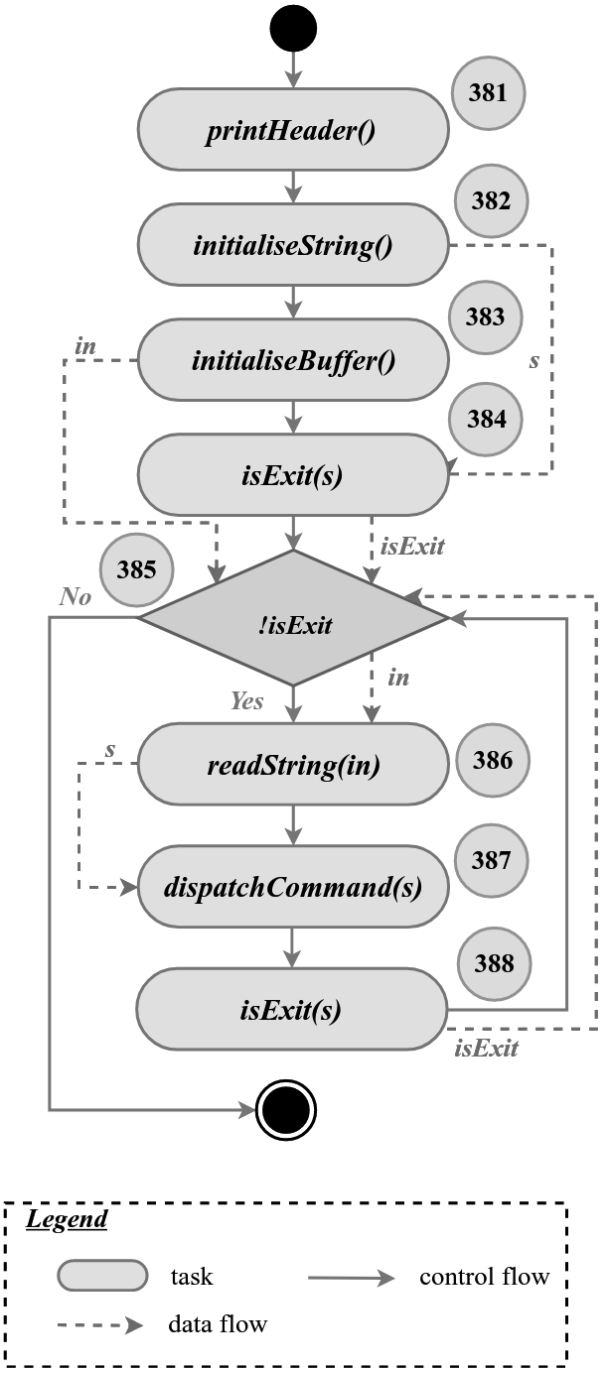

Figure 5.7 – *Workflow extracted from eLib application*

Table 5.14 – *Test cases for eLib application*

Functionality	Test case	Description
Adding a user	addUser Ritedj, 340 rue Ada Bat A Log 21 Montpellier, 0764124578	Adding a new user
	addUser Oussama, 111 rue du Faubourg Bat C Log 50 Montpellier, 0664524971	Adding a new user
Adding an internal user	addIntUser Jack, 340 rue Maurice et Katia Bat A Log 20 Montpellier, 0764264931, 3	Adding a new internal user
	addIntUser Mouaadh, 340 rue Ada Bat F Log 211 Montpellier, 0764264931, 4	Adding a new internal user
Remove a user	rmUser 4	Removing Mouaadh
	rmUser 5	Error: removing unsubscribed user
Adding a book	addBook DevOps, Leonard J. Bass et al, 978-0-1340-4984-7	Add a new book
	addBook Microservices, Eberhard Wolff, 978-0134602417	Add a new book
	addBook Refactoring Databases, Scott W Ambler and Pramod J Sadalage, 978-0321293534	Add a new book
Adding a report	addReport Workflow control-flow patterns: A revised view, BPM-06-22, Nick Russell et al.	Add a new report
Adding a journal	addJournal IEEE Transactions on Software Engineering	Add a new journal
	addJournal International Journal of Software Engineering and Knowledge Engineering	Add a new journal
	addJournal Autoomated Softaware Engineering	Add a new journal
Removing a document	rmDoc 7	Removing the journal "Automated Softaware Engineering"
	rmDoc 8	Error: no document with Id=8
Borrowing a document	borrowDoc 1, 3	Ritedj borrows the book "Refactoring Databases"
	borrowDoc 1,1	Ritedj borrows the book "DevOps"
	borrowDoc 1, 2	Error: Ritedj already has the maximem autorised loans
	borrowDoc 2, 5	Oussama borrows the journal "IEEE Transactions on Software Engineering"
	borrowDoc 3, 3	Error: Jack wants to borrows an unavailable book (already borrowed)
	borrowDoc 4, 3	Error: no user with Id=4
	borrowDoc 1,8	Error: no document with Id=8
Returning a document	returnDoc 5	Oussama returns the journal "IEEE Transactions on Software Engineering"
	returnDoc 6	Error: "International Journal of Software Engineering and Knowledge Engineering" is not borrowed
	returnDoc 8	Error: no document with Id=8
Searching for users	searchUser Ritedj	Searching for the users with the name "Ritedj"
	searchUser Mouaadh	Error: no users with the name "Mouaadh"
Search for documents	searchDoc Microservices	Looking for the documents with the title "Microservices"
	searchDoc Model-Driven Software Migration	Error: no document with this title in the libreary
Finding wheter a user is holding a document	isHolding 1, 3	Finding whether Ritedj is holding the book "Refactoring Databases"
	isHolding 1, 8	Error: no document with Id=8
	isHolding 4, 2	Error: no user with Id=4
Printing Loans	printLoans	Printing all the loans
Printing the information of a document	printDoc 2	Printing the information of the book "Microservices"
	printDoc 8	Error: no document with Id=8
Printing the information of a user	printUser 1	Printing the information of Ritedj
	printUser 4	Error: no user with Id=4

5.2.4 Threats to Validity

Our workflow extraction approach is concerned by two types of threats to validity: internal and external. We will present the main ones in this section.

5.2.4.1 Threats to Internal Validity

Our workflow extraction approach may be affected by the following internal threats:

1. To restructure OO source code, we utilized *Extract method refactoring*. Nevertheless, the statements extracted as new methods are not functionally related. They do not, for instance, manipulate the same variables or perform a well-specified operation. They were extracted as new methods just because they are delimited by user method invocations. Therefore, as a perspective, we intend to restructure the code to obtain methods that have a purpose (e.g., their statements manipulate the same variables to perform a well-specified operation, etc.) [36] [83] [82].

2. Recovering control flow from OO source code statically in the presence of polymorphism and dynamic binding requires considering all the possible run-time types of a receiver. Thus, if a method includes N virtual calls and each one has M run-time types of a receiver, the CFG will contain at least $N*M$ paths. For this reason, we can not guarantee the scalability of the proposed approach. To tackle this problem, we plan to combine dynamic and static analysis. We will utilize dynamic analysis to determine the exact run-time type of a receiver, whereas static analysis collects the remaining information needed by our approach.

3. In our experiments, we recovered CFGs without considering exception-handling. Taking it into account requires extending our control flow recovery method to be able to handle implicit transfers related to exception raising.

4. The presence of aliasing does not affect the applicability of our approach. The only requirement for our data flow recovery in the presence of alias is the availability of some alias analysis [46]. When the list of aliases is available, any definition/use of an alias is considered as a definition/use of all of them. For instance, supposing that variables *var1* and *var2* are aliases. If a task defines *var1*, then its outputs are *var1* and *var2* because any definition of *var1* is a definition of *var2* and vice versa.

5.2.4.2 Threats to External Validity

Our workflow extraction approach could be concerned with the following external threats:

1. The applications utilized in our experimentation were implemented using Java programming language. To apply it on the ones implemented using other languages (e.g., C++, C#, etc.), we have to adapt it to take into consideration the specificities of these languages, such as multiple inheritance in C++.
2. Only three applications have been collected in the experimentation. Therefore, to consolidate and strengthen our approach, we need to experiment it on a large number of case studies, especially those of a considerably large size (i.e., thousands of classes), as well as industrial ones.
3. To validate our approach, we executed the source code corresponding to the workflow and the one corresponding to the analyzed OO application relying on the same test cases. Similarly to any approach using test cases, the principal challenge is determining the ones that cover all the functionalities of an application. We manually identified these test cases based on our sufficient knowledge of the analyzed systems. Thus, we can ensure that the main functionalities are covered.

5.3 Conclusion

In this chapter, the conducted experiments on case studies to validate our approaches have been presented. The obtained results show the relevance of the identified microservices and the correctness of the recovered workflow. In other words, our microservice identification approach proposes an adequate decomposition of an OO application into microservices. Moreover, the recovered workflow by our workflow extraction approach reflects the behavior of the OO application. However, to consolidate and strengthen both approaches, they have to be evaluated on larger applications. Additionally, for microservice identification, a human expert is needed. He/she would provide the recommendations and assess the recovered microservices more objectively.

VI

Conclusion and Future Directions

This chapter presents a summary of the contributions that were proposed in this dissertation. Additionally, it outlines future research directions.

6.1 Summary of Contributions

The ultimate aim of our dissertation is contributing to the migration of monolithic OO applications to microservices in order to adapt them to both cloud and DevOps. Towards this aim, we propose to identify microservices from existing monoliths. Microservice identification is known as a complex and crucial step for a successful migration. Moreover, it can be structure-based or task-based, depending on what does it consider when decomposing a monolith into microservices. As a big picture, on the one hand, we proposed a structure-based identification approach that partitions a monolithic OO application into microservices relying on source code structure, data accesses, and architect recommendations, when available. On the other hand, to ensure the applicability of existing task-based identification approaches, which either suppose that workflows are available or experts can recover their constituents, we proposed an approach that ex-

tracts a workflow from OO source code. More specifically, our contributions are the following:

— **Structure-based identification of microservices from OO source code:** our approach partitions the classes of an OO application into clusters, each one of them represents a microservices. The identification is based on a quality function that assesses the relevance/quality of microservices. Unlike some existing approaches, our function relies on metrics that reflect the "semantics" of the concept "microservice". These metrics are evaluated from the source code based on the relationships between its entities and their dependence on persistent data.

The goal of our approach is to identify microservices with maximized quality function values. For that purpose, two algorithmic models can be used: genetic and clustering algorithms. It is up to the user of our approach to decide which algorithmic model to utilize. The quality function is considered as a similarity measure in the clustering algorithm and as a fitness function in the genetic algorithm.

Besides source code information, our approach uses software architect recommendations, when available, to guide the identification. These recommendations are related mainly to the use of the application and depending on the available ones, different identifications can be carried out.

To validate our approach, it was evaluated qualitatively and quantitatively. The qualitative evaluation was performed relying on three case studies of different sizes. Quantitative evaluation was carried out based on two case studies, and the evaluation results were compared with two state-of-the-art approaches among the well-known ones. The produced results show the relevance of the identified microservices by our approach.

— **Workflow extraction from OO source code:** our approach extracts a workflow based on static analysis of OO source code. This extraction requires the ability to map OO concepts to workflow ones. For that purpose, a mapping model that associates to OO concepts their corresponding in workflows was established. The defined mapping is used by an extraction process consisting of two steps. The first step aims to recover a model of the OO source code. It involves identifying the structural elements of the application and their relationships by analyzing the source code. The second step consists of transforming the OO model into a workflow-based one relying on the defined mapping model. It starts by identifying primitive and composite tasks, as well as their respective input and output data, and then recovering the control flow and the data flow associated to these tasks.

To validate our approach, it was applied on three applications. The obtained results show that the recovered workflow reflects the behavior of the OO application.

6.2 Future Directions

Based on the presented approaches in this dissertation, several future directions have been identified. The main ones are the following:

6.2.1 Addressing Limitations and New Related Aspects

— **Combining static and dynamic analysis:** our approaches rely on static analysis to recover the needed information from the source code. Hence, the problem of handling polymorphism and dynamic binding confronts us. On the one hand, to identify microservices, measuring the quality of each one, and using the clustering algorithm already require considerable computations. Therefore, and since estimating the possible run-time types of the receiving objects of all the virtual calls requires more computations, their static type was used. This did not have a high negative impact on the produced results. Based on the qualitative and quantitative evaluations, our approach identified relevant microservices.

On the other hand, to recover the control flow, we conservatively estimated the run-time type of the receiver. Thus, if a method includes *N* virtual calls and each one have *M* run-time types of a receiver, the CFG will contain at least *N*M* paths. Thus, the scalability of our approach is not assured. To tackle this problem and determine the impact of addressing polymorphism as well as dynamic binding on microservice identification, we plan to combine dynamic and static analysis. Dynamic analysis is used to determine the exact run-time type of a receiver, whereas static analysis collects the remaining information needed by our approaches.

— **Identifying microservices from the extracted workflows:** to identify microservices from the extracted workflows, we intend to either apply the presented approach in [15] or propose our own approach. Once the task-based microservices are identified, they will be compared with the structure-based ones, to determine the impact of taking into account the temporal evolution of execution on microservice identification.

— **Packaging the identified microservices:** our microservice identification approach addresses the first step of the migration process towards microservices. However, to have an operational microservice-based application, the identified microservices need to be packaged. To achieve this purpose, the OO dependencies between them should be transformed. We plan to do that inspired by the proposed approach in [13], which specifies and transforms these dependencies in order to migrate OO applications to component-based one. According to Alshara et al. [13], these dependencies are either explicit or implicit. Explicit ones include method calls and instantiations, whereas

implicit dependencies are caused by inheritance and exception handling. It is noteworthy that even though microservices interfaces have been identified to compare the recovered microservices by our approach with the ones produced by existing approaches, only method calls were handled.

— **Decomposing the database:** our microservice identification approach takes into account the manipulated data by microservices when identifying them. This may facilitate the decomposition of the database, nevertheless it still a challenging task. Several microservices may share database tables. Therefore, an efficient decomposition, aiming to reduce the communications due to data exchanges, may require refactoring the database [14]. We intend to address this problem.

— **Deploying microservices in the cloud and measuring their performance:** since we aim to migrate monolithic applications towards microservices because they are well-adapted to the cloud, we intend to deploy and run them in the cloud using containers, such as Docker. Moreover, we plan to measure the impact of this migration on the performance of applications. Decomposing a monolith into microservices leads to additional communication between them, combined with network latency, they may impact the performance negatively.

— **Extending the scope of software architect recommendations:** our microservice identification approach relies on software architect recommendations, when available. To lighten the expert task as much as possible, and thus reduce its cost, the list of these recommendations was specified. We plan to extend this list in order to identify more relevant microservices. For instance, the architect can specify the set of classes, instead of one class, that constitutes the gravity center of a microservice.

6.2.2 Experimentations and Validations

— **Experimenting on more and larger applications:** both approaches were validated relying on a limited number of applications. Therefore, to consolidate and strengthen them, we intend to experiment on more applications, especially complex and large ones (i.e., thousands of classes). Moreover, we plan to contact industrials aiming to migrate their applications to microservices and collaborate with their software architects to apply our approach. Even though this might be difficult, it still worth trying. Finally, we experimented using only Java applications. To demonstrate that our approaches are not limited just to Java, we plan to adapt them, if necessary, and test them on case studies developed using other OO programming languages such as C++.

— **Combining clustering and genetic algorithms:** the proposed microservice

identification approach enables extracting microservices either based on clustering or genetic algorithms. Our idea is to use both successively by initializing the population of the genetic algorithm relying on the identified microservices by the clustering one, and then applying the genetic algorithm. Once the results are produced, we will evaluate them qualitatively as well as quantitatively, and compare them with those obtained by the current version of our genetic algorithm initialized randomly.

— **Thorough quantitative validation of the identified microservices:** to validate our approach, the quantitative assessment proposed by Jin et al. [76], which evaluate the functional independence of the identified microservices, was used. The authors improved their quantitative evaluation in [77] by considering two other characteristics: modularity and independence of evolvability. If a microservice is well-modularized, its internal entities should be cohesive, whereas external entities should be loosely coupled. The modularity is evaluated relying on an extended version of the proposed modularity measure in [96]. The independence of evolvability is assessed by three metrics: Internal Co-change Frequency (*ICF*), External Co-change Frequency (*ECF*), and Ratio of *ECF* to *ICF* (*REI*). *ICF* and *ECF* are derived from the change history of the monolithic application. We plan to validate our approach relying on the new quantitative evaluation since it considers other characteristics of microservices, and it was applied on a larger number of applications compared to the old one.

Personal Publications

— Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Christophe Dony, and Chouki Tibermacine. Refactoring object-oriented applications for a deployment in the cloud - workflow generation based on static analysis of source code. In *Proceedings of the 13th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2018), Funchal, Madeira, Portugal, March 23-24, 2018.*, pages 111–123, 2018. URL https://doi.org/10.5220/0006699101110123.

— Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, Christophe Dony, and Rahina Oumarou Mahamane. Re-architecting OO software into microservices - A quality-centred approach. In *Proceedings of the the 7th European Conference on Service-Oriented and Cloud Computing (ESOCC 2018), Como, Italy, September 12-14, 2018.*, pages 65–73, 2018. URL https://doi.org/10.1007/978-3-319-99819-0_5.

— Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde-Lilia Bouziane, and Christophe Dony. From object-oriented to workflow: Refactoring of OO applications into workflows for an efficient resources management in the cloud. In *Evaluation of Novel Approaches to Software Engineering - 13th International Conference (ENASE 2018), Funchal, Madeira, Portugal, March 23-24, 2019, Revised Selected Papers*, pages 186–214, 2018. URL https://doi.org/10.1007/978-3-030-22559-9_9.

— Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, and Christophe Dony. From Monolith to Microservice Architectural Style: A Migration Process Based on the Analysis of Source Code Structure, Data Accesses and Architect Recommendations. Being submitted to *Journal of Systemes and Software (JSS)*.

— Rahina Oumarou Mahamane, Anfel Selmadji, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Seza Adjoyan, Hinde Lilia Bouziane, and Christophe Dony. Microservice Identification Using a Multi-Objective Genetic Algorithm. To be submitted to *European Conference on Software Architecture (ECSA) 2020* or *International Conference on Software Architecture (ICSA) 2020*.

Appendices

A

Call Graph Construction Algorithms Based on Static Analysis of OO Source Code

Our workflow extraction approach identifies tasks and recovers control as well as data flows based on static analysis of OO source code. To identify tasks, we relied on the call graph of the analyzed OO application. In literature, several algorithms [49, 20, 21, 132, 129] have been proposed to construct call graphs by statically analyzing OO source code. According to Tip and Palsberg [132], the common idea between these algorithms is to abstract an object (resp., a set of objects) into the name of its class (resp., a set of their class names). Then, for a given method call *obj.m()*, the goal is to determine a set of class names that approximate the run-time types of the receiver *obj*. Once this set is computed, the possible invoked methods can be determined by examining the class hierarchy. Several well-known call graph construction algorithms will be presented in this appendix.

Reachability Analysis (RA)

RA is a simple algorithm for constructing a call graph. It only considers method names. The basic idea is to determine for each method m, the names of its callees. Then create an edge between m and all the methods that have the same name of any callee. A slightly more advanced version of this algorithm takes into account method signatures instead of their names.

Figure A.1 shows a call graph constructed using RA. In this example, since the method *Main.m1* (resp., *Main.m2*) contains a virtual call *param1.m()* (resp., *obj3.m()*), edges from the node representing *Main.m1* (resp., *Main.m2*) towards all the nodes corresponding to methods with the name *m* were created.

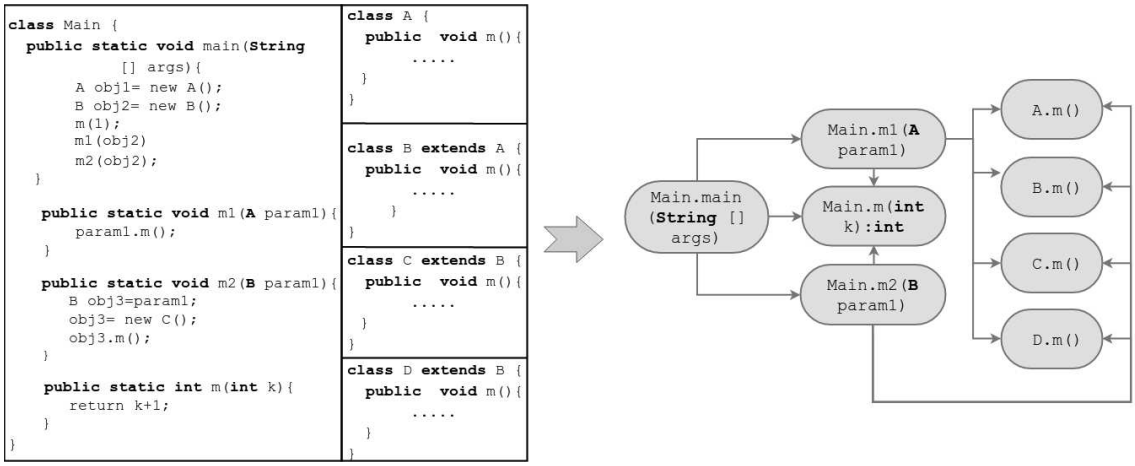

Figure A.1 – *Example of a call graph constructed using RA*

Class Hierarchy Analysis (CHA)

The call graph constructed using RA can be improved by considering the class hierarchy. For that purpose, CHA [49] have been proposed. This analysis uses the static type of a receiver *obj* of a virtual call *obj.m()*, combined with the class hierarchy, to determine the methods that can be invoked at run-time. Every method with the name *m* that is inherited by a subtype of the static type of *obj* is a possible target.

An example is shown in Figure A.2. CHA removed three edges from the call graph built using RA.

Rapid Type Analysis (RTA)

RTA [20, 21] is an improvement of CHA by taking into account class instantiation information. A possible type of a receiver can only be a class that has been instantiated in the OO application via a *new*. Therefore, by collecting class instantiations, it is possible to improve results produced by CHA. More precisely, given a receiver *obj* of a virtual call *obj.m()* within an OO application, the approximated run-time types of *obj* are the classes belonging to the intersection between its class hierarchy and the instantiated classes in this application. Note that the call graph is constructed on-the-fly, and only the instantiations that are in the methods already contained in this graph are taken into account.

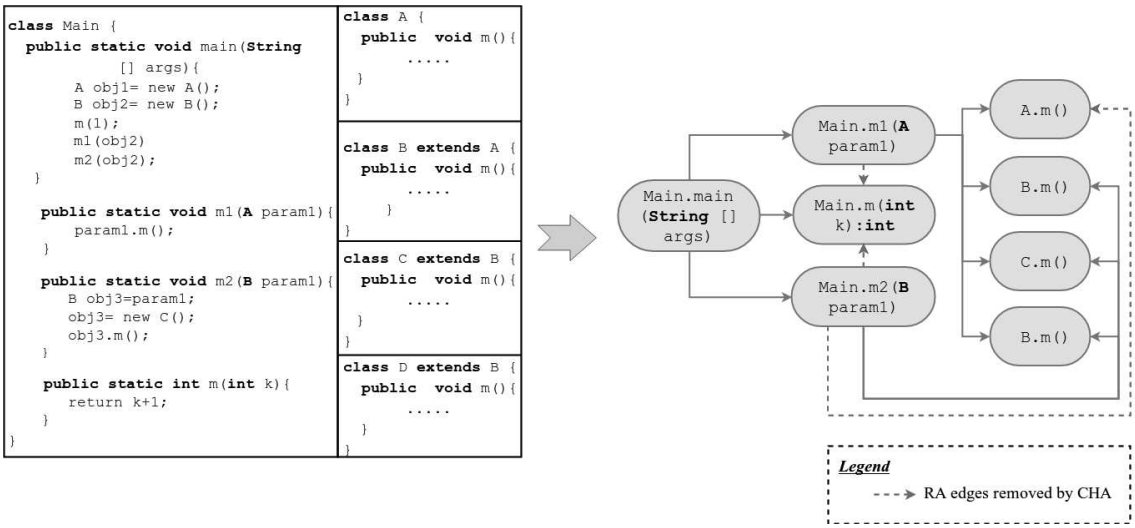

Figure A.2 – *Example of a call graph constructed using CHA*

Figure A.3 shows an example of a call graph produced by RTA. Two of the edges constructed by CHA were removed.

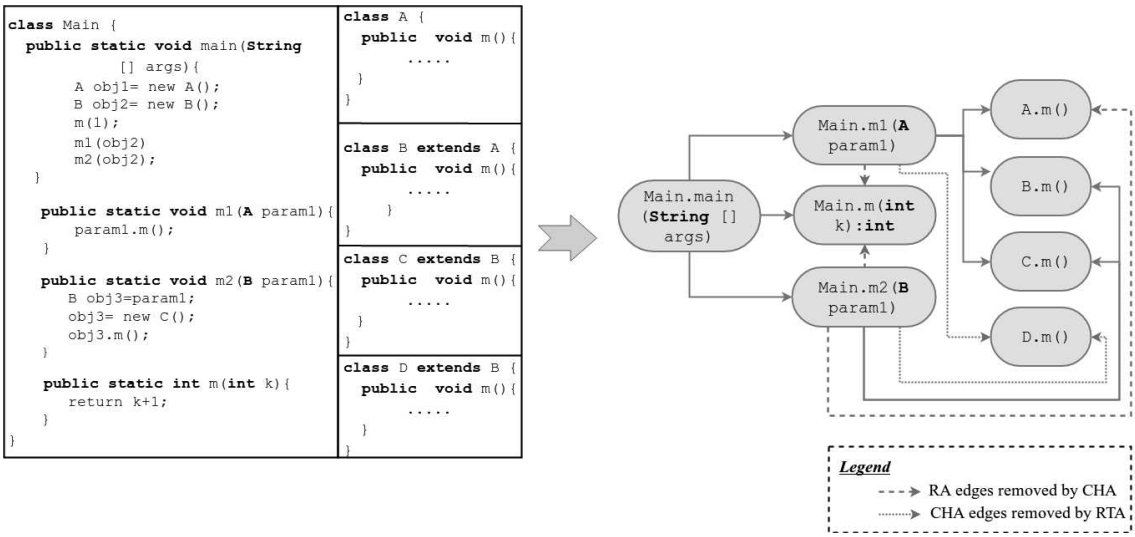

Figure A.3 – *Example of a call graph constructed using RTA*

XTA Analysis

XTA [132] is an improvement of RTA. The idea is that by giving methods and fields a more precise local view of the types of objects available, virtual calls may be resolved more precisely. Therefore, instead of considering a single set of class instantiations for the entire application, a local set is defined for each method S_m

and each field S_x.

To compute these sets, Tip and Palsberg [132] defined the following constraints:

1. $main \in R$
2. For each method m, each virtual call $obj.m1()$ in m, and each class $C \in SubTypes\ (StaticType(obj))$ where $StaticLookup(C, m1) = m'$:
   $(m \in R) \wedge (C \in S_m) \implies$
   $$\begin{cases} m' \in R \wedge \\ SubTypes(ParamTypes(m')) \cap S_m \subseteq S_{m'} \wedge \\ SubTypes(ReturnType(m')) \cap S_{m'} \subseteq S_m \wedge \\ C \in S_{m'} \end{cases}$$
3. For each method m, and for each "$newC()$" in m:
   $(m \in R) \implies C \in S_m$
4. For each method m reading a field x:
   $(m \in R) \implies S_x \subseteq S_m$
5. For each method m writing a field x:
   $(m \in R) \implies (SubTypes(StaticType(x)) \cap S_m) \subseteq S_x$

Where:

— R represents the set of reachable methods.
— $StaticType(obj)$ is the static type of obj.
— $SubTypes(t)$ denotes the set of declared subtypes of t.
— $StaticLookup(C, m)$ represents a definition of a method named m that can be found in the class C by a static method lookup.
— $ParamTypes(m)$ the static types of the parameters of m.
— $ReturnedType(m)$ the returned type by m.

The first constraint supposes that the $main$ method of the OO application is reachable. The second constraint captures the flow of data from m to m', and from m' back to m through the two inclusions. The third constraint adds the class C to the set of instantiations available in m. The fourth constraint expresses the data flow from a field to a method. Whereas, the fifth constraint reflects a data flow from a method to a field, while taking into account class hierarchy, as well as instantiation information.

Figure A.4 shows an example of a call graph built by XTA. One edge constructed by RTA was removed.

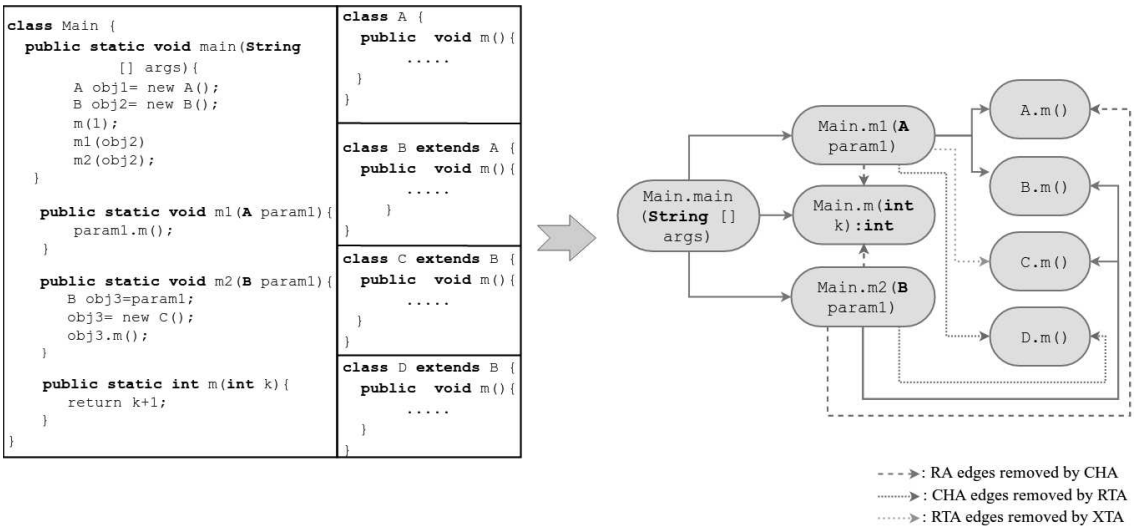

Figure A.4 – *Example of a call graph constructed using XTA*

Variants of XTA

By associating a distinct set to each class, method and/or field, different variants of XTA can be obtained. In [132], three of them have been experimented:

— *Class Type Analysis (CTA):* in CTA, a distinct set S_C is used for each class C.

— *Method Type Analysis (MTA):* MTA associates a distinct set S_C to each class C and a set S_x to each field x.

— *Field Type Analysis (FTA):* FTA associates a distinct set S_C to each class C and a set S_m to each method m.

Variable Type Analysis (VTA)

VTA [129] is an enhancement of RTA. Indeed, instead of collecting class instantiations from the OO application and use them to prune the call graph, VTA relies on the types of objects that can reach each variable (i.e., the instantiations that might be assigned to this variable). VTA is based on a type propagation graph. In this graph, nodes represent variables, whereas edges specify the flow of types due to assignments. Note that even implicit assignments, resulting from method invocations and returns, are taken into account. To build a call graph, VTA proceeds by:

1. Building the type propagation graph.

2. Initializing reaching type information using assignments of the form $obj = newC()$ (i.e., the type C is associated to the node representing obj).
3. Propagating type information along directed edges to find out the set of types reaching each variable.
4. Filtering out impossible reaching types by intersecting with possible types as indicated in the class hierarchy (i.e., $ReachingTypes(obj) \cap SubTypes(obj)$).

In order that the analysis be simple, VTA was implemented based on a typed 3-address representation which provides explicit names and types for all local variables.

An example of a call graph constructed using VTA is shown in Figure A.5. One edge from the call graph constructed by XTA was removed.

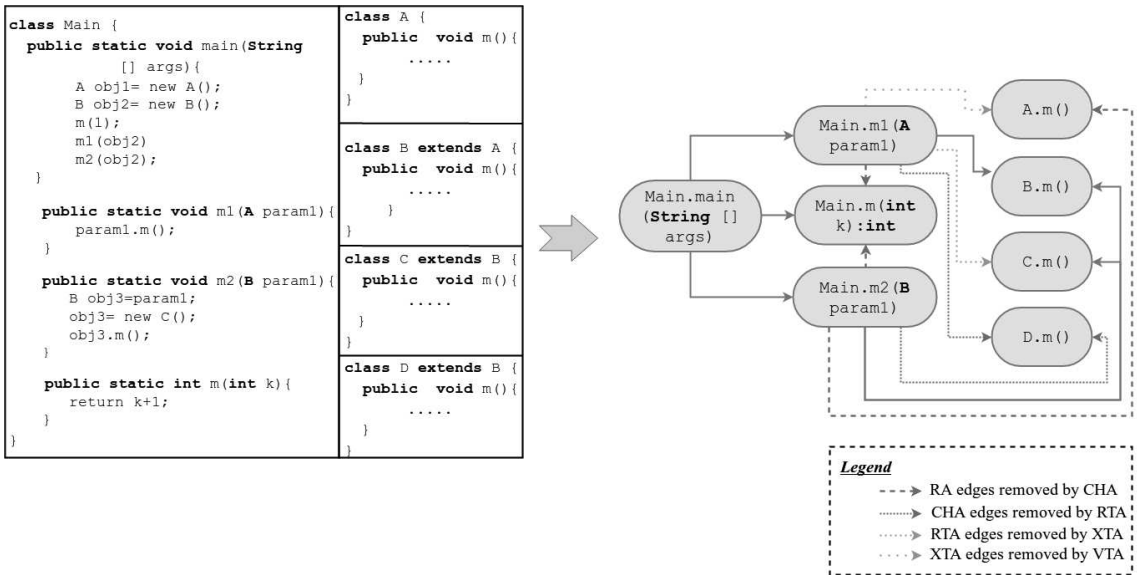

Figure A.5 – *Example of a call graph constructed using VTA*

Bibliography

[1] Domain Object Model. https://www.w3.org/DOM/. Accessed: May 2019. 45

[2] Google App Engine. https://cloud.google.com/appengine. Accessed: May 2019. x, 2

[3] Amazon EC2. https://aws.amazon.com/ec2/. Accessed: May 2019. x, 2

[4] Microsoft Azure. https://azure.microsoft.com. Accessed: May 2019. x, 2

[5] Salesforce. https://www.salesforce.com/au/saas/. Accessed: May 2019. x, 2

[6] ISO/IEC 25010:2011, systems and software engineering - systems and software quality requirements and evaluation (SQuaRE) - system and software quality models. Technical report, British Standards Institution, 2013. URL https://pdfs.semanticscholar.org/57a5/b99eceff9da205e244337c9f4678b5b23d25.pdf. 24, 63

[7] Seza Adjoyan. *Describing Dynamic and Variable Software Architecture Based on Identified Services From Object-Oriented Legacy Applications. (Architecture Dynamique Basée sur la Description de la Variabilité et des Services Identifiés Depuis des Applications Orientées Objet)*. PhD thesis, University of Montpellier, France, 2016. URL https://tel.archives-ouvertes.fr/tel-01693061. 17, 18, 32

[8] Seza Adjoyan, Abdelhak-Djamel Seriai, and Anas Shatnawi. Service identification based on quality metrics: Object-oriented legacy system migration towards SOA. In *The 26th International Conference on Software Engineering and Knowledge Engineering, Hyatt Regency, Vancouver, BC, Canada, July 1-3, 2013.*, pages 1–6, 2014. 17, 18, 123

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World

student series edition. Addison-Wesley, 1986. ISBN 0-201-10088-6. URL http://www.worldcat.org/oclc/12285707. 115

[10] Zakarea Al-Shara. *Migrating Object Oriented Applications into Component-Based ones (Migration des Applications Orientées Objet vers Celles á Base de Composants)*. PhD thesis, University of Montpellier, France, 2016. URL https://tel.archives-ouvertes.fr/tel-01816975/. 14, 15, 26, 32, 33

[11] Simon Allier, Houari A. Sahraoui, Salah Sadou, and Stéphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *Component-Based Software Engineering, 13th International Symposium, CBSE 2010, Prague, Czech Republic, June 23-25, 2010. Proceedings*, pages 216–231, 2010. URL https://doi.org/10.1007/978-3-642-13238-4_13. 18

[12] Simon Allier, Salah Sadou, Houari A. Sahraoui, and Régis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011, Boulder, Colorado, USA, June 20-24, 2011*, pages 214–223, 2011. URL https://doi.org/10.1109/WICSA.2011.35. 17, 18

[13] Zakarea Alshara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde-Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Migrating large object-oriented applications into component-based ones: Instantiation and inheritance transformation. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2015, Pittsburgh, PA, USA, October 26-27, 2015*, pages 55–64, 2015. URL https://doi.org/10.1145/2814204.2814223. 18, 155

[14] Scott W Ambler and Pramod J Sadalage. *Refactoring Databases: Evolutionary Database Design*. Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, 2006. ISBN 978-0321293534. 156

[15] Mohammad Javad Amiri. Object-aware identification of microservices. In *2018 IEEE International Conference on Services Computing, SCC 2018, San Francisco, CA, USA, July 2-7, 2018*, pages 253–256, 2018. URL https://doi.org/10.1109/SCC.2018.00042. xiv, xvi, 9, 10, 19, 22, 25, 26, 28, 29, 31, 34, 35, 36, 38, 74, 155

[16] Vasilios Andrikopoulos, Tobias Binz, Frank Leymann, and Steve Strauch. How to adapt applications for the cloud environment: Challenges and solutions in migrating applications to the cloud. *Computing*, 95(6):493–535, 2013. URL https://doi.org/10.1007/s00607-012-0248-2. x, 2

[17] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010. URL http://doi.acm.org/10.1145/1721654.1721672. x, 2

[18] Dionysis Athanasopoulos. Usage-aware service identification for architecture migration of object-oriented systems to soa. In *Database and Expert Systems Applications - 28th International Conference, DEXA 2017, Lyon, France, August 28-31, 2017, Proceedings, Part II*, pages 54–64, 2017. URL https://doi.org/10.1007/978-3-319-64471-4_6. 18

[19] Dionysis Athanasopoulos, Apostolos V. Zarras, George Miskos, Valérie Issarny, and Panos Vassiliadis. Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Transactions on Services Computing*, 8(4):550–562, 2015. URL https://doi.org/10.1109/TSC.2014.2310195. 121, 122

[20] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, USA, October 6-10, 1996.*, pages 324–341, 1996. URL https://doi.org/10.1145/236337.236371. 105, 163, 164

[21] David Francis Bacon. *Fast and effective optimization of statically typed object-oriented languages*. PhD thesis, University of California, Berkeley, USA, 1997. URL http://web.cs.ucla.edu/~palsberg/tba/papers/bacon-thesis97.pdf. 105, 163, 164

[22] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report. In *Advances in Service-Oriented and Cloud Computing - Workshops of ESOCC 2015, Taormina, Italy, September 15-17, 2015, Revised Selected Papers*, pages 201–215, 2015. URL https://doi.org/10.1007/978-3-319-33313-7_15. x, xi, 3, 4

[23] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016. URL https://doi.org/10.1109/MS.2016.64. x, 2, 19, 57

[24] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian A. Tamburri, and Theo Lynn. Microservices migration patterns. *Softw., Pract. Exper.*, 48(11):2019–2042, 2018. URL https://doi.org/10.1002/spe.2608. 73

[25] Luciano Baresi, Martin Garriga, and Alan De Renzis. Microservices identification through interface analysis. In *Service-Oriented and Cloud Computing - 6th IFIP WG 2.14 European Conference, ESOCC 2017, Oslo, Norway, September 27-29, 2017, Proceedings*, pages 19–33, 2017. URL https://doi.org/10.1007/978-3-319-67262-5_2. xiii, 8, 19, 20, 26, 28, 30, 31, 33, 34, 35, 36, 38

[26] Leonard J. Bass, Ingo M. Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. SEI series in software engineering. Addison-Wesley, 2015. ISBN 978-0-1340-4984-7. URL https://www.oreilly.com/library/view/devops-a-software/9780134049885/. x, 2

[27] James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *ACM SIGSOFT Symposium on Software Reusability (SSR)*, pages 259–262, 1995. URL https://doi.org/10.1145/211782.211856. 71

[28] Dominik Birkmeier and Sven Overhage. On component identification approaches: Classification, state of the art, and comparison. In *Component-Based Software Engineering, 12th International Symposium, CBSE 2009, East Stroudsburg, PA, USA, June 24-26, 2009, Proceedings*, pages 1–18, 2009. URL https://doi.org/10.1007/978-3-642-02414-6_1. 18

[29] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information system migration: A brief review of problems, solutions and research issues. Technical report, Computer Science Department, Trinity College, Dublin, Ireland, 1999. URL https://pdfs.semanticscholar.org/5481/704b2c7a1e92d44b8583c26fb3e3cd6096b5.pdf. 14

[30] Jesus Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, 1999. URL https://doi.org/10.1109/52.795108. xiii, 7

[31] Antonio Bucchiarone, Nicola Dragoni, Schahram Dustdar, Stephan Thordal Larsen, and Manuel Mazzara. From monolithic to microservices: An experience report from the banking domain. *IEEE Software*, 35(3):50–55, 2018. URL https://doi.org/10.1109/MS.2018.2141026. 19, 57

[32] Georg Buchgeher, Mario Winterer, Rainer Weinreich, Johannes Luger, Roland Wingelhofer, and Mario Aistleitner. Microservices in a small development organization - an industrial experience report. In *Software Architecture - 11th European Conference, ECSA 2017, Canterbury, UK, September 11-15, 2017, Proceedings*, pages 208–215, 2017. URL https://doi.org/10.1007/978-3-319-65831-5_15. 19

[33] Ugo A. Buy, Alessandro Orso, and Mauro Pezzè. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2000, Portland, OR, USA, August 21-24, 2000*, pages 39–48, 2000. URL https://doi.org/10.1145/347324.348870. xiv, 9, 19, 42, 43, 45, 48, 49, 50, 51, 52, 54, 55, 56

[34] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Comp. Syst.*, 25(6):599–616, 2009. URL https://doi.org/10.1016/j.future.2008.12.001. x, 2

[35] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: technologies, development frameworks, and quality assurance schemes. In *7th Asia-Pacific Software Engineering Conference (APSEC 2000), 5-8 December 2000, Singapore*, page 372, 2000. URL https://doi.org/10.1109/APSEC.2000.896722. 17

[36] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, Antonios Gkortzis, and Paris Avgeriou. Identifying extract method refactoring opportunities based on functional relevance. *IEEE Transactions on Software Engineering*, 43(10):954–974, 2017. URL https://doi.org/10.1109/TSE.2016.2645572. 150

[37] Sylvain Chardigny. *Extraction d'une architecture logicielle à base de composants depuis un système orienté objet: Une aproche par exploration. (Component-based software architecture recovery from an object oriented system: A search-based approach)*. PhD thesis, University of Nantes, France, 2009. URL https://tel.archives-ouvertes.fr/tel-00456367. 67, 88, 89, 140

[38] Sylvain Chardigny, Abdelhak Seriai, Mourad Oussalah, and Dalila Tamzalit. Extraction of component-based architecture from object-oriented systems. In *Seventh Working IEEE / IFIP Conference on Software Architecture (WICSA 2008), 18-22 February 2008, Vancouver, BC, Canada*, pages 285–288, 2008. URL https://doi.org/10.1109/WICSA.2008.44. 18

[39] Feng Chen, Shaoyun Li, and William Cheng-Chung Chu. Feature analysis for service-oriented reengineering. In *12th Asia-Pacific Software Engineering Conference (APSEC 2005), 15-17 December 2005, Taipei, Taiwan*, pages 201–208, 2005. URL https://doi.org/10.1109/APSEC.2005.67. 18

[40] Feng Chen, Zhuopeng Zhang, Jianzhi Li, Jian Kang, and Hongji Yang. Service identification via ontology mapping. In *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009), Seattle, Washington, USA, July 20-24, 2009. Volume 1*, pages 486–491, 2009. URL https://doi.org/10.1109/COMPSAC.2009.71. 18

[41] Lianping Chen. Continuous delivery: Huge benefits, but challenges too. *IEEE Software*, 32(2):50–54, 2015. URL https://doi.org/10.1109/MS.2015.27. x, 3

[42] Lianping Chen. Microservices: Architecting for continuous delivery and devops. In *IEEE International Conference on Software Architecture (ICSA 2018), Seattle, WA, USA, April 30 - May 4, 2018*, pages 39–46, 2018. URL https://doi.org/10.1109/ICSA.2018.00013. 19, 57

[43] Mei-Hwa Chen and Howard M. Kao. Testing object-oriented programs: An integrated approach. In *10th International Symposium on Software Reliability Engineering, ISSRE, 1999, Boca Raton, FL, USA, November 1-4, 1999*, pages 73–82, 1999. URL https://doi.org/10.1109/ISSRE.1999.809312. xiv, 9, 19, 42, 43, 45, 48, 49, 50, 51, 52, 54, 55, 56, 108

[44] Rui Chen, Shanshan Li, and Zheng Li. From monolith to microservices: A dataflow-driven approach. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, pages 466–475, 2017. URL https://doi.org/10.1109/APSEC.2017.53. xiii, xiv, xvi, 8, 9, 10, 19, 22, 25, 26, 27, 28, 31, 32, 34, 35, 36, 38

[45] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. URL https://doi.org/10.1109/52.43044. 14, 15, 16

[46] Dave Clarke, James Noble, and Tobias Wrigstad, editors. *Aliasing in Object-Oriented Programming: Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013. ISBN 978-3-642-36945-2. URL https://doi.org/10.1007/978-3-642-36946-9. 150

[47] Brad J. Cox and Andrew J. Novobilski. *Object-oriented programming: An evolutionary approach*. Addison-Wesley, 1991. ISBN 978-0-201-54834-1. 17

[48] Charles Darwin. *On the Origin of Species: A facsimile of the first edition*. Harvard Paperbacks series. Harvard University Press, 2001. ISBN 978-0674637528. 67

[49] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP'95), Århus, Denmark, August 7-11, 1995, Proceedings*, pages 77–101, 1995. URL https://doi.org/10.1007/3-540-49538-X_5. 105, 163, 164

[50] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002. URL https://doi.org/10.1109/4235.996017. 30

[51] Tharam Dillon, Chen Wu, and Elizabeth Chang. Cloud computing: Issues and challenges. In *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, pages 27–33. Ieee, 2010. x, 2

[52] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering*, pages 195–216. Springer, 2017. URL https://doi.org/10.1007/978-3-319-67425-4_12. xi, xii, 4, 5, 6, 24

[53] Stéphane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35 (4):573–591, 2009. URL https://doi.org/10.1109/TSE.2009.19. 18

[54] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *ICSE Workshop on Dynamic Analysis (WODA 2003). New Mexico State University Portland, OR*, pages 24–27, 2003. URL https://homes.cs.washington.edu/~mernst/pubs/staticdynamic-woda2003.pdf. 32

[55] Javier Espadas, Arturo Molina, Guillermo Jiménez, Martín Molina, Raúl Ramírez, and David Concha. A tenant-based resource allocation model for scaling software-as-a-service applications over cloud computing infrastructures. *Future Generation Computer Systems*, 29(1):273–286, 2013. URL https://doi.org/10.1016/j.future.2011.10.013. ix, 2

[56] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. Challenges in delivering software in the cloud as microservices. *IEEE Cloud Computing*, 3(5):10–14, 2016. URL https://doi.org/10.1109/MCC.2016.105. xi, 4

[57] Fairouz Fakhfakh, Hatem Hadj Kacem, and Ahmed Hadj Kacem. Workflow scheduling in cloud computing: A survey. In *18th IEEE International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOC Workshops 2014), Ulm, Germany, September 1-2, 2014*, pages 372–378, 2014. URL https://doi.org/10.1109/EDOCW.2014.61. x, 2

[58] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN 978-0-201-48567-7. URL http://martinfowler.com/books/refactoring.html. 15, 16

[59] Martin Fowler. Continuous Integration. https://martinfowler.com/articles/continuousIntegration.html, 2006. Accessed: May 2019. x, 3

[60] Andrei Furda, Colin J. Fidge, Olaf Zimmermann, Wayne Kelly, and Alistair Barros. Migrating enterprise legacy source code to microservices: On multitenancy, statefulness, and data consistency. *IEEE Software*, 35(3):63–72, 2018. URL https://doi.org/10.1109/MS.2017.440134612. 19

[61] Walid Gaaloul and Claude Godart. Mining workflow recovery from event based logs. In *Business Process Management, 3rd International Conference, BPM 2005, Nancy, France, September 5-8, 2005, Proceedings*, pages 169–185, 2005. URL https://doi.org/10.1007/11538394_12. xiv, 9, 19, 42, 43, 44, 45, 48, 50, 51, 52, 54, 55

[62] Walid Gaaloul, Khaled Gaaloul, Sami Bhiri, Armin Haller, and Manfred Hauswirth. Log-based transactional workflow mining. *Distributed and Parallel Databases*, 25(3):193–240, 2009. URL https://doi.org/10.1007/s10619-009-7040-0. xiv, 9, 15, 19, 42, 43, 44, 45, 48, 50, 51, 52, 54, 55

[63] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison Wesley, 1994. ISBN 978-0201633610. 46

[64] David García Gil and Rubén Aguilera Díaz-Heredero. A microservices experience in the banking industry. In *Proceedings of the 12th European Conference on Software Architecture (ECSA 2018): Companion Proceedings, Madrid, Spain, September 24-28, 2018*, pages 13:1–13:2, 2018. URL https://doi.org/10.1145/3241403.3241418. 19

[65] Jean-Philippe Gouigoux and Dalila Tamzalit. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops, ICSA Workshops 2017, Gothenburg, Sweden, April 5-7, 2017*, pages 62–65, 2017. URL https://doi.org/10.1109/ICSAW.2017.35. 19, 57

[66] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: A systematic approach to service decomposition. In *Service-Oriented and Cloud Computing - 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, pages 185–200, 2016. URL https://doi.org/10.1007/978-3-319-44482-6_12. xiii, 8, 19, 20, 26, 27, 28, 31, 32, 34, 35, 36, 38

[67] Mark Harman. Why source code analysis and manipulation will always be important. In *Tenth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2010), Timisoara, Romania, 12-13 September 2010*, pages 7–19, 2010. URL https://doi.org/10.1109/SCAM.2010.28. 25

[68] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979. URL https://www.jstor.org/stable/2346830?seq=1#page_scan_tab_contents. 67

[69] John H Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence*. University of Michigan Press, 1975. ISBN 978-0472084609. URL https://books.google.fr/books/about/Adaptation_in_natural_and_artificial_sys.html?id=JE5RAAAAMAAJ&redir_esc=y. xvi, 10, 28, 67

[70] D Hollingsworth. Workflow management coalition: The workflow reference model. Technical report, Workflow Management Coalition, 1995. 29, 38

[71] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build*. Addison Wesley, 2010. ISBN 978-0321601919. x, 3

[72] Nacim Ihaddadene. Extraction of business process models from workflow events logs. *International Journal of Parallel, Emergent and Distributed Systems*, 23(3):247–258, 2008. URL https://doi.org/10.1080/17445760701536183. xiv, 9, 15, 19, 42, 43, 44, 45, 48, 50, 51, 52, 54, 55

[73] Anil K Jain, Richard C Dubes, et al. *Algorithms for clustering data*. Prentice Hall Advanced Refrence Series. Prentice hall Englewood Cliffs, 1988. ISBN 0-13-022278-X. URL https://homepages.inf.ed.ac.uk/rbf/BOOKS/JAIN/Clustering_Jain_Dubes.pdf. 66

[74] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Cloud migration research: A systematic review. *IEEE Transactions on Cloud Computing*, 1(2): 142–157, 2013. URL https://doi.org/10.1109/TCC.2013.10. x, 2

[75] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35, 2018. URL https://doi.org/10.1109/MS.2018.2141039. 18

[76] Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *2018 IEEE International Conference on Web Services (ICWS 2018), San Francisco, CA, USA, July 2-7, 2018*, pages 211–218, 2018. URL https://doi.org/10.1109/ICWS.2018.00034. 19, 20, 26, 27, 28, 31, 32, 34, 35, 36, 38, 118, 120, 121, 123, 135, 143, 157

[77] Wuxia Jin, Ting Liu, Yuanfang Cai, Rick Kazman, Ran Mo, and Qinghua Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE Transactions on Software Engineering*, 2019. URL https://ieeexplore.ieee.org/document/8686152. 15, 19, 20, 25, 26, 27, 28, 30, 31, 32, 33, 34, 35, 36, 38, 157

[78] Prasad Jogalekar and C. Murray Woodside. Evaluating the scalability of distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 11(6):589–603, 2000. URL https://doi.org/10.1109/71.862209. 24

[79] Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32 (3):241–254, 1967. URL https://link.springer.com/article/10.1007/BF02289588. 60, 79

[80] Hui Kang, Michael Le, and Shu Tao. Container and microservice driven design for cloud infrastructure devops. In *2016 IEEE International Conference on Cloud Engineering (IC2E 2016), Berlin, Germany, April 4-8, 2016*, pages 202–211, 2016. URL https://doi.org/10.1109/IC2E.2016.26. 25

[81] Dimka Karastoyanova and Alejandro P. Buchmann. Components, middleware and web services. In *Proceedings of the IADIS International Conference WWW/Internet (ICWI 2003), Algarve, Portugal, November 5-8, 2003*, pages 967–970, 2003. 17

[82] Mehmet Kaya and James W. Fawcett. Identifying extract method opportunities based on variable references (S). In *The 25th International Conference on Software Engineering and Knowledge Engineering, Boston, MA, USA, June 27-29, 2013.*, pages 153–158, 2013. 150

[83] Mehmet Kaya and James W. Fawcett. Identification of extract method refactoring opportunities through analysis of variable declarations and uses. *International Journal of Software Engineering and Knowledge Engineering*, 27(1): 49–70, 2017. URL https://doi.org/10.1142/S0218194017500036. 150

[84] Rick Kazman, Steven S. Woods, and S. Jeromy Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *5th Working Conference on Reverse Engineering, WCRE '98, Honolulu, Hawai, USA, October 12-14, 1998*, pages 154–163, 1998. URL https://doi.org/10.1109/WCRE.1998.723185. 14, 15

[85] Selim Kebir, Abdelhak-Djamel Seriai, Sylvain Chardigny, and Allaoua Chaoui. Quality-centric approach for software component identification from object-oriented code. In *2012 Joint Working IEEE/IFIP Conference on*

Software Architecture and European Conference on Software Architecture (WICSA/ECSA 2012), Helsinki, Finland, August 20-24, 2012, pages 181–190, 2012. URL https://doi.org/10.1109/WICSA-ECSA.212.26. 18

[86] Gabor Kecskemeti, Attila Csaba Marosi, and Attila Kertész. The ENTICE approach to decompose monolithic services into microservices. In *International Conference on High Performance Computing & Simulation (HPCS 2016), Innsbruck, Austria, July 18-22, 2016*, pages 591–596, 2016. URL https://doi.org/10.1109/HPCSim.2016.7568389. 19, 20, 26, 27, 28, 30, 31, 34, 35, 36, 38

[87] Holger Knoche and Wilhelm Hasselbring. Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49, 2018. URL https://doi.org/10.1109/MS.2018.2141035. 19

[88] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006. URL https://doi.org/10.1016/j.ress.2005.11.018. 67, 68, 89

[89] E. Korshunova, Marija Petkovic, M. G. J. van den Brand, and Mohammad Reza Mousavi. CPP2XMI: reverse engineering of UML class, sequence, and activity diagrams from C++ source code. In *13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy*, pages 297–298, 2006. URL https://doi.org/10.1109/WCRE.2006.21. xiv, 9, 19, 43, 45, 50, 51, 52, 54, 55

[90] David A. Kosower and Juan J. Lopez-Villarejo. Flowgen: Flowchart-based documentation for C++ codes. *Computer Physics Communications*, 196:497–505, 2015. URL https://doi.org/10.1016/j.cpc.2015.05.029. xiv, 9, 19, 41, 43, 44, 45, 48, 49, 50, 51, 52, 54, 55, 56

[91] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007. URL https://doi.org/10.1109/TSE.2007.70726. 17

[92] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. URL https://ieeexplore.ieee.org/document/1456074. 24

[93] Timothy Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20 (6):35–39, 2003. URL https://doi.org/10.1109/MS.2003.1241364. xiii, 7

[94] Alessandra Levcovitz, Ricardo Terra, and Marco Tulio Valente. Towards a technique for extracting microservices from monolithic enterprise systems. *CoRR*, abs/1605.03175, 2016. URL http://arxiv.org/abs/1605.03175. xiii, 8, 19, 20, 24, 26, 28, 29, 31, 34, 35, 36, 38

[95] James Lewis and Martin Fowler. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html, 2014. Accessed: May 2019. xi, xii, 4, 5, 17, 63, 64, 72

[96] Spiros Mancoridis, Brian S. Mitchell, Chris Rorres, Yih-Farn Chen, and Emden R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*, pages 45–52, 1998. URL https://doi.org/10.1109/WPC.1998.693283. 157

[97] Andrian Marcus and Jonathan I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*, pages 125–137, 2003. URL https://doi.org/10.1109/ICSE.2003.1201194. 41

[98] Andrian Marcus and Denys Poshyvanyk. The conceptual cohesion of classes. In *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*, pages 133–142, 2005. URL https://doi.org/10.1109/ICSM.2005.89. 32, 33

[99] Vincenzo Martena, Alessandro Orso, and Mauro Pezzè. Interclass testing of object oriented software. In *8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), 2-4 December 2002, Greenbelt, MD, USA*, pages 135–144, 2002. URL https://doi.org/10.1109/ICECCS.2002.1181506. xiv, 9, 19, 42, 43, 45, 48, 49, 50, 51, 52, 54, 55, 56, 108

[100] Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *IEEE International Conference on Web Services (ICWS 2017), Honolulu, HI, USA, June 25-30, 2017*, pages 524–531, 2017. URL https://doi.org/10.1109/ICWS.2017.61. xiii, 8, 18, 19, 20, 25, 26, 27, 28, 29, 31, 33, 34, 35, 36, 38, 118, 135

[101] Peter Mell, Tim Grance, et al. The NIST definition of cloud computing. *National Institute of Standards and Technology, Gaithersburg*, 2011. ix, x, 2

[102] Brian S. Mitchell, Martin Traverso, and Spiros Mancoridis. An architecture for distributing the computation of software clustering algorithms. In *Working IEEE / IFIP Conference on Software Architecture (WICSA 2001), 28-31 August 2001, Amsterdam, The Netherlands*, pages 181–190, 2001. URL https://doi.org/10.1109/WICSA.2001.948427. 29

[103] Melanie Mitchell. *An introduction to genetic algorithms*. MIT Press, 1998. ISBN 978-0-262-63185-3. xvi, 10, 28

[104] Tadahiko Murata, Hisao Ishibuchi, and Hideo Tanaka. Multi-objective genetic algorithm and its applications to flowshop scheduling. *Computers & Industrial Engineering*, 30(4):957–968, 1996. URL https://www.sciencedirect.com/science/article/pii/0360835296000459. 60, 90, 91

[105] Iulian Neamtiu and Tudor Dumitras. Cloud software upgrades: Challenges and opportunities. In *5th IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011), Williamsburg, VA, USA, September 26, 2011*, pages 1–10, 2011. URL https://doi.org/10.1109/MESOCA.2011.6049037. x, 2

[106] Sam Newman. *Building microservices: Designing fine-grained systems, 1st Edition*. O'Reilly, 2015. ISBN 9781491950357. URL http://www.worldcat.org/oclc/904463848. xi, 4, 17, 57, 63, 64, 72, 120

[107] Michael P. Papazoglou. *Web Services: Principles and Technology*. Prentice Hall, 2008. ISBN 978-0-321-15555-9. 17

[108] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: Approaches, technologies and research issues. *VLDB J.*, 16(3):389–415, 2007. URL https://doi.org/10.1007/s00778-007-0044-3. 17

[109] Enric Plaza. On reusing other people's experiences. *Künstliche Intelligenz (KI)*, 23(1):18–23, 2009. 42

[110] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010. ISBN 978-3-642-12577-5. URL http://www.springer.com/computer/swe/book/978-3-642-12577-5?changeHeader. 26, 44

[111] Denys Poshyvanyk and Andrian Marcus. The conceptual coupling metrics for object-oriented systems. In *22nd IEEE International Conference on Software Maintenance (ICSM 2006), 24-27 September 2006, Philadelphia, Pennsylvania, USA*, pages 469–478, 2006. URL https://doi.org/10.1109/ICSM.2006.67. 32, 33

[112] Chris Richardson. Pattern: Monolithic Architecture. https://microservices.io/patterns/monolithic.html, . Accessed: May 2019. xii, 5, 24

[113] Chris Richardson. Pattern: Microservice Architecture. https://microservices.io/patterns/microservices.html, . Accessed: May 2019. 6

[114] Chris Richardson. *Microservice patterns*. Manning Publications, 2017. ISBN 978-1617294549. xi, xii, 4, 5, 17, 18

[115] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, 1991. ISBN 0-13-629841-9. URL https://dl.acm.org/citation.cfm?id=130437. 25

[116] Nick Russell, Arthur HM Ter Hofstede, Wil MP Van Der Aalst, and Nataliya Mulyar. Workflow control-flow patterns: A revised view. *BPM Center Report BPM-06-22*, 2006. 106

[117] Dragan Savic. Single-objective vs. multiobjective optimisation for integrated decision support. In *Proceedings of the International Congress of Environmental Modelling and Software (iEMSs)*, pages 7–12, 2002. URL https://scholarsarchive.byu.edu/iemssconference/2002/all/119/. 90

[118] Pol Schumacher, Mirjam Minor, Kirstin Walter, and Ralph Bergmann. Extraction of procedural knowledge from the web: a comparison of two workflow extraction approaches. In *Proceedings of the 21st World Wide Web Conference (WWW), Lyon, France, April 16-20, 2012 (Companion Volume)*, pages 739–747, 2012. URL https://doi.org/10.1145/2187980.2188194. xiv, 9, 19, 42, 43, 44, 45, 46, 48, 50, 51, 52, 54, 55, 56

[119] Matthias Schur, Andreas Roth, and Andreas Zeller. Mining workflow models from web applications. *IEEE Transactions on Software Engineering*, 41(12):1184–1201, 2015. URL https://doi.org/10.1109/TSE.2015.2461542. xiv, xxi, 9, 19, 41, 42, 43, 44, 45, 47, 48, 50, 51, 52, 54, 55, 56

[120] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI series in software engineering. Addison-Wesley, 2003. ISBN 978-0-321-11884-4. URL http://www.informit.com/store/modernizing-legacy-systems-software-technologies-engineering-9780321118844. 14

[121] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017. URL https://ieeexplore.ieee.org/document/7884954. x, 3

[122] Sourabh Sharma. *Mastering Microservices with Java*. Packt Publishing Limited, 2016. ISBN 978-1785285172. xi, xii, 4, 5, 17, 63, 64

[123] Sourabh Sharma, Rajech RV, and David Gonzalez. *Microservices: Building scalable software*. Packt Publishing, 2017. ISBN 1787280985. URL https://www.packtpub.com/application-development/microservices-building-scalable-software. xi, 4, 17, 64

[124] Anas Shatnawi. *Supporting Reuse by Reverse Engineering Software Architecture and Component from Object-Oriented Product Variants and APIs . (Support à la réutilisation par la rétro-ingénierie des architectures et des composants logiciels à partir du code source orienté objet des variantes de produits logiciels et d'APIs)*. PhD thesis, University of Montpellier, France, 2015. URL https://tel.archives-ouvertes.fr/tel-01322864. 17, 20, 25, 27, 30, 31, 32

[125] Anas Shatnawi, Hafedh Mili, Manel Abdellatif, Ghizlane El-Boussaidi, Yann-Gaël Guéhéneuc, Naouel Moha, and Jean Privat. What should you know before developing a service identification approach. *CoRR*, abs/1803.05282, 2018. URL http://arxiv.org/abs/1803.05282. 18

[126] Harry M. Sneed. Integrating legacy software into a service oriented architecture. In *10th European Conference on Software Maintenance and Reengineering (CSMR 2006), 22-24 March 2006, Bari, Italy*, pages 3–14, 2006. URL https://doi.org/10.1109/CSMR.2006.28. 18

[127] Maria Alejandra Rodriguez Sossa. *Resource provisioning and scheduling algorithms for scientific workflows in cloud computing environments*. PhD thesis, University of Melbourne, Australia, 2016. URL http://hdl.handle.net/11343/119597. ix, x, 2

[128] Rod Stephens. *Beginning software engineering*. Wrox, 2015. ISBN 978-1118969144. xii, 5

[129] Vijay Sundaresan, Laurie J. Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. Practical virtual method call resolution for java. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 264–280, 2000. URL https://doi.org/10.1145/353171.353189. 105, 106, 163, 167

[130] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017. URL https://doi.org/10.1109/MCC.2017.4250931. xi, 4, 5, 24

[131] Bill Thomas and Scott R. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *The Nineteenth Annual International Conference of Computer Documentation: Communicating in the*

New Millennium (SIGDOC 2001), Santa Fe, New Mexico, USA, October 21-24, 2001, pages 235–236, 2001. URL https://doi.org/10.1145/501516.501570. 41

[132] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2000), Minneapolis, Minnesota, USA, October 15-19, 2000.*, pages 281–293, 2000. 105, 106, 163, 165, 166, 167

[133] Paolo Tonella and Alessandra Potrich. *Reverse Engineering of Object Oriented Code*. Monographs in Computer Science. Springer, 2005. ISBN 978-0-387-40295-6. URL https://doi.org/10.1007/b102522. 143

[134] John R Vacca. *Computer and information security handbook*. Morgan Kaufmann, 2017. ISBN 978-0123943972. x, 3

[135] Wil M. P. van der Aalst, Boudewijn F. van Dongen, Joachim Herbst, Laura Maruster, Guido Schimm, and A. J. M. M. Weijters. Workflow mining: A survey of issues and approaches. *Data & Knowledge Engineering*, 47(2): 237–267, 2003. URL https://doi.org/10.1016/S0169-023X(03)00066-1. 44

[136] Anneliese von Mayrhauser and A. Marie Vans. Program understanding: Models and experiments. *Advances in Computers*, 40:1–38, 1995. URL https://doi.org/10.1016/S0065-2458(08)60543-4. 24

[137] Christian Wagner. *Model-Driven Software Migration: A Methodology - Reengineering, Recovery and Modernization of Legacy Systems*. Springer, 2014. ISBN 978-3-658-05269-0. URL https://doi.org/10.1007/978-3-658-05270-6. 14, 65

[138] Dolores R. Wallace and Roger U. Fujii. Software verification and validation: An overview. *IEEE Software*, 6(3):10–17, 1989. URL https://doi.org/10.1109/52.28119. 42

[139] Dolores R Wallace, Laura M Ippolito, and Barbara B Cuthill. *Reference information for the software verification and validation process*. Diane Pub Co, 1996. ISBN 978-0788143403. 42

[140] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Science of Computer Programming*, 56(1-2):99–116, 2005. URL https://doi.org/10.1016/j.scico.2004.11.007. 17, 18

[141] Eberhard Wolff. *Microservices: Flexible Software Architecture*. Addison Wesley, 2016. ISBN 978-0134602417. URL https://www.worldcat.org/title/microservices-flexible-software-architecture/oclc/965730846&referer=brief_results. 24

[142] Rui Xu and Donald C. Wunsch II. Survey of clustering algorithms. *IEEE Trans. Neural Networks*, 16(3):645–678, 2005. URL https://doi.org/10.1109/TNN.2005.845141. xvi, 10, 29, 66, 67

[143] Edward Yourdon and Larry L Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc., 1979. 27

[144] Zhuopeng Zhang and Hongji Yang. Incubating services in legacy systems for architectural migration. In *11th Asia-Pacific Software Engineering Conference (APSEC 2004), 30 November - 3 December 2004, Busan, Korea*, pages 196–203, 2004. URL https://doi.org/10.1109/APSEC.2004.61. 18

[145] Zhuopeng Zhang, Hongji Yang, and William C. Chu. Extracting reusable object-oriented legacy code segments with combined formal concept analysis and slicing techniques for service integration. In *Sixth International Conference on Quality Software (QSIC 2006), 26-28 October 2006, Beijing, China*, pages 385–392, 2006. URL https://doi.org/10.1109/QSIC.2006.29. 18

[146] Liming Zhu, Len Bass, and George Champlin-Scharff. Devops and its practices. *IEEE Software*, 33(3):32–34, 2016. URL https://doi.org/10.1109/MS.2016.81. xii, 6, 25

[147] Ying Zou and Maokeng Hung. An approach for extracting workflows from e-commerce applications. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pages 127–136, 2006. URL https://doi.org/10.1109/ICPC.2006.9. xiv, 9, 19, 43, 45, 48, 50, 51, 52, 54, 55

[148] Ying Zou, Terence C. Lau, Kostas Kontogiannis, Tack Tong, and Ross McKegney. Model-driven business process recovery. In *11th Working Conference on Reverse Engineering (WCRE 2004), Delft, The Netherlands, November 8-12, 2004*, pages 224–233, 2004. URL https://doi.org/10.1109/WCRE.2004.30. xiv, 9, 15, 19, 42, 43, 45, 46, 48, 50, 51, 52, 54, 55, 56