Evolution of microservice-based applications : Modelling and safe dynamic updating

Yuwei Wang

▶ To cite this version:

Yuwei Wang. Evolution of microservice-based applications : Modelling and safe dynamic updating. Artificial Intelligence [cs.AI]. Institut Polytechnique de Paris, 2022. English. NNT : 2022IPPAS009 . tel-03937239

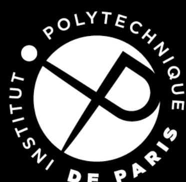
HAL Id: tel-03937239

https://theses.hal.science/tel-03937239

Submitted on 13 Jan 2023

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

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



INSTITUT
POLYTECHNIQUE
DE PARIS

NNT : 2022IPPAS009

Thèse de doctorat

Evolution of Microservice-based Applications: Modelling and Safe Dynamic Updating

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom SudParis

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Informatique

Thèse présentée et soutenue à Palaiseau, France, le **27 October 2022**, par

Yuwei Wang

Composition du Jury :

Thomas Ledoux Professeur, IMT Atlantique (laboratoire LS2N)	Président
Antoine Beugnard Professeur, IMT Atlantique (laboratoire Lab-STICC)	Rapporteur
Lionel Seinturier Professeur, Université de Lille (laboratoire Cristal)	Rapporteur
Fabienne Boyer Maître de conférences (HDR), Université de Grenoble (laboratoire LIG)	Examinatrice
Sophie Chabridon Directrice d'Études, IPP/TSP (laboratoire SAMOVAR)	Directrice de thèse
Denis Conan Maître de conférences (HDR), IPP/TSP (laboratoire SAMOVAR)	Co-encadrant de thèse
Kavoos Bojnourdi Ingénieur de recherche, EDF R&D	Co-encadrant (Invité)
Jingxuan Ma Ingénieur de recherche, EDF R&D	Co-encadrante (Invitée)

Acknowledgements

First and foremost, I would like to express my deep gratitude to my research supervisors, Sophie Chabridon and Denis Conan, for their guidance to help me throughout my PhD study. They steered me to the road of research thanks to their patience, motivation, and immense knowledge. I could not have imagined having better advisors and mentors for my PhD study. My sincere thanks also goes to my industry advisors, Kavoos Bojnourdi and Jingxuan Ma, for sharing their expertise and giving me valuable guidance during my PhD study.

I would like to thank Prof. Antoine Beugnard and Prof. Lionel Seinturier for spending time to review my thesis. Their insightful comments and suggestions during the mini-defense helped me to improve my thesis and widen my research from various perspectives. I would also like to thank my other thesis committee members, Prof. Thomas Ledoux and Prof. Fabienne Boyer.

In addition, I would like to mention Viktor Colas, Tsimafei Liashkevich and Julio Guzmán Barraza for their contributions to the implementation of prototypes.

I take this opportunity to express gratitude to all the members of the ACMES group of SAMOVAR lab and the I2A group of EDF for a cherished time spent together with me in the lab and in the company. Many thanks to everyone in the Computer Science department at Télécom Sudparis and the Seido Lab at EDF, particularly to Alexei Mikchevitch for his help in administrative matters.

I am also grateful to my friends for their time, advice and moral support, and for all the fun we have had over the past few years. Thanks to Silun Zhang for giving me the strength and hope, and Yifang Dong for our seven years of studying abroad experience in France together.

Last but not least, I would like to thank my parents, Guihong Chen and Hongbiao Wang, for their unconditional love and encouragement in every moment of my life. As well as my grandfathers Dingchun Chen and Wentai Wang who accompany me from heaven. None of this could have happened without the great support of my family. Special thanks to my dog Point for all the entertainment and emotional support.

Abstract

Microservice architectures contribute to building complex distributed systems as sets of independent microservices. The decoupling and modularity of distributed microservices facilitates their independent replacement and upgradeability. Since the emergence of agile DevOps and CI/CD, there is a trend towards more frequent and rapid evolutionary changes of the running microservice-based applications in response to various evolution requirements. Applying changes to microservice architectures is performed by an evolution process of moving from the current application version to a new version. The maintenance and evolution costs of these distributed systems increase rapidly with the number of microservices.

The objective of this thesis is to address the following issues: How to help engineers to build a unified and efficient version management for microservices and how to trace changes in microservice-based applications? When can microservice-based applications, especially those with long-running activities, be dynamically updated without stopping the execution of the whole system, and how should the safe updating be performed to ensure service continuity and maintain system consistency?

In response to these questions, this thesis proposes two main contributions. The first contribution is runtime models and an evolution graph for modelling and tracing version management of microservices, which are built at design time and used at runtime. It helps engineers abstract architectural evolution in order to manage reconfiguration deployments, and it provides the knowledge base to be manipulated by an autonomic manager middleware in various evolution activities. The second contribution is a snapshot-based approach for dynamic software updating (DSU) of microservices. The consistent distributed snapshots of running microservice-based application are constructed to be used for specifying continuity of service, evaluating the safe update conditions and realising the update strategies. The message complexity of the DSU algorithm is not the message complexity of the distributed application, but the complexity of the consistent distributed snapshot algorithm.

Keywords: Microservice architecture; Software evolution; Version management; Model at runtime; Dynamic software updating; Snapshot-based update condition detection.

Synopsis en Français

Ce synopsis est fourni en conformité avec la loi de 1994 relative à l'emploi de la langue française. Il reprend la structure de la thèse et résume les chapitres un à un.

Chapitre 1:Introduction

Les architectures à base de microservices contribuent à la construction de systèmes répartis complexes sous forme d'ensembles de microservices indépendants. La modularité des microservices facilite leur remplacement de manière indépendante et leur mise à niveau, ce qui permet une approche *DevOps* agile et une évolution continue. La modélisation et la gestion des changements évolutifs rapides des applications basées sur les microservices constituent la base des solutions permettant d'effectuer des mises à jour dynamiques de ces applications avec un temps d'arrêt minimal.

Cette thèse étant soutenue par une Convention industrielle de formation par la recherche (Cifre) en coopération avec EDF, nous avons identifié plusieurs besoins industriels, liés à l'évolution, insuffisamment pris en compte jusqu'à présent :

— en raison de l'hétérogénéité des microservices dans l'écosystème de l'entreprise, une représentation globale et unifiée des architectures de microservices devrait être fournie aux architectes, développeurs et administrateurs ;

— l'évolution du logiciel pour les architectures de microservices devrait être modélisée et suivre des règles bien définies lorsqu'un microservice change, ou que certaines dépendances de microservices changent ;

— le système doit être capable de gérer les versions obsolètes des microservices afin de libérer des ressources inutiles ;

— le système doit permettre la traçabilité des changements évolutifs du système et rendre possible le passage à une version spécifique ou à une version précédente ;

— le système doit fournir un service continu et minimiser autant que possible la durée des interruptions du client. Il ne doit pas non plus bloquer les parties du système non affectées par les reconfigurations d'un ensemble donné de microservices.

Nous répondons à une partie de ces besoins dans nos deux contributions visant, d'une part, à aider les ingénieures et les ingénieurs à réaliser des activités de modélisation et de gestion de versions pour les microservices, et d'autre part, à améliorer la continuité de service lors de la mise à jour dynamique des applications basées sur les microservices.

Un prototype en logiciel libre été réalisé pour valider la première contribution : MIMOSAE (pour "MIcroservices MOdel for verSion mAnagement with Evolution graph"), https://gitlabev.imtbs-tsp.eu/mimosae/mimosae. Une version préliminaire d'un prototype mettant en œuvre la seconde contribution est également disponible : ARBORE (pour "ARchitecting Based on microservice versiOns with REconfiguration"), https://gitlabev.imtbs-tsp.eu/mimosae/arbore.

Chapitre 2 : État de l'art sur l'évolution logicielle dynamique des architectures microservices

L'architecture logicielle, l'une des disciplines du génie logiciel, fournit un schéma directeur pour les systèmes logiciels complexes. Elle fournit une abstraction de haut niveau de la structure globale, du comportement et des propriétés des systèmes logiciels.

Les architectures logicielles ont évolué depuis les architectures monolithiques vers différentes granularités de modularité : des objets aux composants et services, et maintenant aux microservices. La modularité de l'architecture logicielle décrit le degré auquel les éléments architecturaux peuvent être séparés et recombinés dans le but de créer et de maintenir des modules exécutables indépendamment afin de maximiser la réutilisation de leur code.

Les architectures logicielles des applications d'entreprise deviennent de plus en plus grandes et complexes, et de nombreux efforts ont été faits pour répondre à cette augmentation d'échelle. Les architectures à base de microservice sont l'un des modèles architecturaux qui permet de concevoir, de développer et de déployer des systèmes évolutifs et flexibles.

L'évolution des logiciels correspond au processus de développement, de maintenance et de mise à jour des logiciels tout au long de leur cycle de vie. Les modifications et mises à jour sont inévitables dans le développement des logiciels afin de répondre à l'évolution des exigences, des performances, etc. Un domaine de recherche actif concerne la manière de gérer et de contrôler les tâches d'évolution en limitant l'intervention humaine et en

minimisant les durées d'arrêt. Dans cette thèse, nous nous concentrons sur l'évolution dynamique qui permet au système logiciel de s'adapter aux changements en cours d'exécution et nous nous appuyons sur deux approches : la boucle de contrôle MAPE-K pour l'automatisation et l'approche *modèle-à-l'exécution* pour refléter l'architecture de l'application à faire évoluer.

L'approche MAPE-K permet d'automatiser ces tâches en proposant un modèle générique d'une boucle de contrôle autonomique. Un gestionnaire autonomique organise le processus de contrôle autour de quatre activités : "Surveiller", "Analyser", "Planifier" et "Exécuter" qui partagent une "Base de connaissances". L'activité de surveillance collecte les données des éléments gérés et de leur environnement. L'activité d'analyse détermine s'il y a lieu de procéder à une adaptation en utilisant les informations provenant de l'activité de surveillance et de la base de connaissances. L'activité de planification reçoit les décisions d'adaptation et détermine les actions de configuration ou de déploiement nécessaires. L'activité d'exécution applique le plan d'adaptation au système géré aux moments appropriés. La base de connaissances maintient et partage les règles, les propriétés et les modèles. Notre travail est applicable à l'adaptation semi-automatique où une partie du processus de contrôle est réalisée par les ingénieurs et les ingénieures, et une autre partie par le gestionnaire autonomique.

Un modèle d'exécution fournit une représentation réduite des éléments logiciels hétérogènes qui sont disponibles dans le référentiel d'implémentation et des éléments gérés de l'application en cours d'exécution. Les adaptations logicielles sont appliquées aux modèles maintenus au moment de l'exécution avant d'être exécutées dans les applications gérées. L'abstraction de haut niveau de l'application à travers les modèles d'exécution fournit une vue unifiée au gestionnaire autonomique et l'aide à construire la base de connaissances partagée par la boucle de contrôle.

Les architectures de microservices mettent l'accent sur l'évolutivité, car la modularité des microservices permet d'évoluer de manière indépendante et rend les microservices plus faciles à modifier et à remplacer. Cependant, la mise à jour dynamique correcte des architectures de microservices est encore un domaine de recherche ouvert et constitue l'objet de cette thèse.

La mise à jour des microservices à l'exécution doit répondre à plusieurs exigences, telles que la continuité de service, la correction, l'intégrité et la cohérence du système. Dans notre travail, nous faisons les hypothèses suivantes : (i) nous ignorons les transferts d'états nécessaires entre les versions des microservices, et (ii) notre système géré est sans défaillance.

Afin de caractériser les changements, nous suivons la politique de SemVer pour exprimer les numéros de version des éléments qui doivent être versionnés dans les

architectures de microservices. Nous complétons SemVer avec la classification des évolutions en changements essentiels ou non essentiels en tenant compte des aspects comportementaux du modèle du microservice.

Les solutions DSU (pour *Dynamic Software Updating* en anglais) de l'état de l'art, qui répondent à la question de savoir quand une mise à jour peut être effectuée de manière correcte et cohérente, présentent des caractéristiques communes. Toutes ces solutions visent en effet la cohérence globale (de l'ensemble de l'application répartie) et traitent la cohérence du système par l'évitement, c'est-à-dire en attendant un état sûr (condition de mise à jour) avant de procéder à la mise à jour. Notre solution suit également cette approche, mais elle se différencie par la manière d'atteindre et de maintenir la condition de mise à jour, indiquant lorsque le système peut être mis à jour en toute sécurité. Tous les algorithmes DSU existants dans la littérature sont dits basés sur la dissémination, alors que notre contribution propose une nouvelle approche basée sur les instantanés (ou *snapshots*).

Chapitre 3 : Modèles d'exécution et graphe d'évolution pour la gestion des versions des microservices

Ce chapitre présente notre première contribution : la construction de modèles d'exécution et de graphes d'évolution pour (i) aider les ingénieures et ingénieurs à gérer la gestion des versions d'applications de microservices, et (ii) abstraire l'évolution architecturale afin de gérer les déploiements de reconfiguration. Plus précisément, ces besoins peuvent être affinés en fonction des besoins typiques suivants identifiés dans des contextes industriels : (1) afin de rationaliser le coût d'évolution et de maintenance de solutions logicielles hétérogènes, une vue globale et unifiée de l'évolution de l'écosystème doit être fournie aux architectes, développeurs et administrateurs, et (2) les changements évolutifs du système doivent être traçables et annulés en cas de configuration invalide ou de toute autre anomalie.

Notre modélisation sépare la vue des types de celle des instances, ajoute la gestion des versions à chaque élément du modèle et prend en compte les modes de communication synchrones et asynchrones. Notre modèle est réifié au moment de l'exécution pour faire partie de la base de connaissances de la boucle de contrôle MAPE-K. Les éléments gérés sont alors reflétés dans le modèle d'exécution. Avec l'évolution de l'application microservice, si des changements se produisent dans l'application, les modèles changent également, d'abord le modèle de type puis le modèle d'instance, et vice versa.

Notre graphe d'évolution est utilisé pour suivre la trajectoire d'évolution de l'architecture de microservices. Chaque fois que les artefacts d'un type de microservice sont

ajoutés ou retirés du référentiel d'implémentation, le modèle de type peut évoluer pour prendre en compte l'évolution, et un nouveau nœud de type est créé et engagé dans le graphe d'évolution. La validation n'est possible que si les types de microservices sont instanciables. Le nouveau nœud représente l'ensemble des artefacts logiciels (microservices, connecteurs, contrats, etc.) qui peuvent être utilisés pour construire le système géré. La deuxième partie du graphe d'évolution correspond aux instantanés des configurations déployées. La validation n'est possible que si les instances de microservices sont déployables. Chaque fois qu'une décision est prise par les architectes logiciels, un nouveau nœud d'instance est créé et engagé dans le graphe d'évolution. Un tel un nœud représente l'ensemble des entités déployées (instances déployées de microservices, connecteurs, etc.) lorsque le plan de reconfiguration est calculé et exécuté pour modifier efficacement l'application répartie. Enfin, notre solution est mise en œuvre dans le prototype MIMOSAE ("MIcroservices MOdel for verSion mAnagement with Evolution graph") disponible sur https://gitlabev.imtbs-tsp.eu/mimosae/mimosae. Lorsqu'une nouvelle configuration à déployer est inscrite dans le graphe d'évolution, un planificateur calcule un plan d'actions de déploiement pour faire évoluer le système vers cette nouvelle configuration. Ce plan est automatiquement exécuté pour effectuer le déploiement.

Chapitre 4 : Mise à jour logicielle dynamique des microservices basée sur les instantanés

Ce chapitre présente notre deuxième contribution, à savoir quand et comment la reconfiguration pour la mise à jour des microservices peut être effectuée de manière cohérente alors que les appels de services clients continuent d'arriver. Ce problème de mise à jour dynamique du logiciel (DSU) peut être décomposé en deux sous-problèmes à résoudre dans l'ordre : (i) modéliser l'application répartie et les changements de version, et exprimer la condition de mise à jour dans ce modèle de manière à ce que la mise à jour ne conduise pas à des incohérences sémantiques : par exemple, quels éléments architecturaux de l'application répartie évoluent et doivent évoluer ensemble, qu'est-ce qu'un appel de service client, et les microservices peuvent-ils être mis à jour au milieu d'un appel de service client ? (ii) spécifier l'algorithme DSU qui surveille le système en cours d'exécution pour atteindre la condition de mise à jour et qui effectue ensuite la mise à jour en suivant une stratégie donnée : par exemple, le système permet-il aux deux versions d'un microservice de s'exécuter simultanément, l'algorithme DSU doit-il bloquer certains messages ou certains éléments architecturaux ?

Nous abordons le problème de la mise à jour dynamique du logiciel au niveau de la

configuration pour les applications basées sur les microservices. En utilisant le modèle d'exécution, le service intergiciel de gestion autonomique décide quand la mise à jour des microservices peut être effectuée de manière correcte et comment les microservices impactés sont maintenus dans un état cohérent. En outre, un tel service intergiciel peut utiliser différentes stratégies de mise à jour afin de trouver un compromis entre la caractérisation de l'impact des changements (par exemple, s'ils sont essentiels ou non) et la minimisation de la durée des interruptions. L'application répartie évolue ensuite de manière incrémentale d'une configuration à une autre.

Les algorithmes DSU existants dans la littérature sont basés sur la diffusion : que ce soit en période hors mise à jour ou pendant la mise à jour, ils complètent l'application avec des messages de contrôle pour détecter la condition de mise à jour. Ainsi, ces algorithmes sont optimaux dans le pire des cas. Pour les architectures à base de microservices, nous adoptons une approche différente : seulement de manière périodique ou lorsque la condition de mise à jour doit être vérifiée, par exemple lorsqu'il y a une demande de reconfiguration due à des changements de version, notre algorithme DSU prend un instantané global périodique de l'application répartie. La justification de ce choix est la suivante : (1) les applications de microservices peuvent être gourmandes en messages, de sorte que cibler la complexité des messages dans le pire des cas est limitatif, (2) la propriété bidirectionnelle des liens ne peut être obtenue de manière directe avec les connecteurs de type *publier-souscrire*, et il n'est pas raisonnable de supposer que les courtiers sont configurés à cette fin, (3) il est possible d'exprimer la condition de mise à jour à l'aide d'états globaux cohérents du système, c'est-à-dire d'instantanés répartis cohérents, et (4) l'algorithme DSU, qui utilise des primitives telles que la suspension/reprise, l'exécution, la création et la suppression de microservices, la liaison et la dissociation de microservices, etc., est classiquement basé sur des phases, qui sont délimitées par des états globaux du système, c'est-à-dire des instantanés répartis.

Enfin, il existe des situations dans lesquelles une entité est spécifiquement conçue pour être composée avec d'autres, et les deux entités doivent évoluer simultanément. Certains travaux antérieurs ont abordé ce problème, mais uniquement du point de vue de l'implémentation. Nous considérons la situation au moment de la spécification, et nous formalisons la propriété de continuité du service. En outre, la connaissance des ensembles de remplacement pour exprimer la continuité du service devrait également intéresser les administrateurs de systèmes afin de permettre des stratégies de déploiement telles que le déploiement *blue/green* ou l'approche *Canari*.

Notre solution DSU est mise en oeuvre dans une version préliminaire du prototype ARBORE ("ARchitecting basée sur les versions de microservices avec REconfiguration"), disponible sur `https://gitlabev.imtbs-tsp.eu/mimosae/arbore`.

Chapitre 5 : Conclusion

Nos contributions visent à aider les ingénieures et les ingénieurs à gérer plus facilement et plus efficacement la gestion des versions des applications à base de microservices, et à garantir une mise à jour dynamique sûre des logiciels. Nous résumons ci-dessous ce qui fait l'originalité des solutions que nous proposons.

Spécification et modélisation d'une application basée sur des microservices pour la gestion de versions. Le modèle de type décrit l'abstraction structurelle des architectures de microservices, et le modèle d'instance capture les configurations de déploiement spécifiques des applications basées sur les microservices. Chaque instance est conforme à un type valide. Dans le modèle de type, nous vérifions que les types de microservices sont instanciables. Dans le modèle d'instance, nous vérifions que les instances de microservices sont déployables. Ces modèles sont construits au moment de la conception par les ingénieurs et utilisés au moment de l'exécution pour refléter les changements qui se produisent dans l'application, selon l'approche *modèle-à-l'exécution*. Les éléments architecturaux à versionner sont fournis dans le modèle de type avec un identifiant et un numéro de version, ce qui suit explicitement la politique SemVer appropriée sur le plan syntaxique. Ensuite, les éléments déployables dans le modèle d'instance qui instancient les types sont également versionnés syntaxiquement. Nos modèles d'exécution prennent également en charge deux mécanismes de communication couramment utilisés dans les architectures de microservices, à savoir la communication synchrone client-serveur et la communication asynchrone de type *publier-souscrire*.

Tracer les changements évolutifs dans les architectures de microservices. Un graphe d'évolution enregistre la trajectoire et l'historique de l'évolution de l'application dans le temps. Il est composé de deux parties : la première est constituée d'instantanés de types de configuration construits à partir du modèle de type, et la seconde est constituée d'instantanés d'instances de configuration construits à partir du modèle d'instance. Lorsque des changements de version sont appliqués, de nouveaux instantanés du type de configuration et de l'instance de configuration sont créés et enregistrés dans notre graphe d'évolution. Chaque instantané d'instance de configuration est conforme à un instantané de type de configuration. Dans le prototype MIMOSAE, une boucle de contrôle MAPE-K semi-automatique est mise en œuvre dans laquelle nos modèles servent de base de connaissances. Un planificateur IA est utilisé pour calculer un plan contenant un ensemble d'actions à exécuter pour passer d'une configuration donnée à une configuration cible.

Formulation de la mise à jour dynamique basée sur des instantanés. Un modèle d'exécution de microservices est mis en place pour indiquer les liens de dépendance entre les microservices représentant les relations d'appel entre les microservices dans une configuration, et les dépendances entre les types de messages qui sont utilisées pour suivre la progression des échanges de messages tout en répondant aux appels des clients. Ensuite, les changements de microservices sont caractérisés comme étant essentiels ou non essentiels en utilisant ce modèle d'exécution : un changement est essentiel s'il a un impact sur les échanges de messages. En ce qui concerne la continuité de service, nous introduisons deux ensembles de remplacement (paires d'instances de microservices et paires de liens de communication), et nous ajoutons une façade (*front-end*) entre les clients et les microservices afin de passer facilement des instances de l'ancienne version aux instances de la nouvelle version. Ensuite, avec l'introduction du concept d'instantané réparti cohérent, nous formalisons la correction d'une mise à jour dynamique ainsi que les conditions de mise à jour de la littérature (quiescence, *freeness* [en anglais], qui permet la cohérence des versions, et *freeness* essentielle, qui prend en compte les changements essentiels et non essentiels). En conséquence, il est possible d'évaluer la condition de mise à jour uniquement en cas de besoin, ou périodiquement, en prenant des photos (au lieu de compléter systématiquement tous les échanges applicatifs entre microservices par des messages de contrôle). Ceci explique pourquoi notre deuxième contribution est nommée d'après le concept de *snapshot* ou instantané.

Algorithme DSU par vague. L'algorithme DSU assure la correction des mises à jour dynamiques en préservant l'achèvement correct des appels de service client (collaborations) en cours et futurs. La base de l'algorithme DSU est un algorithme de détection de terminaison répartie qui repose sur des instantanés répartis cohérents. Les microservices comptent les messages applicatifs qui sont envoyés et reçus pour chaque collaboration, et fournissent ces compteurs dans le cadre de leur instantané. Ensuite, le gestionnaire autonomique prend un instantané réparti cohérent et détecte la terminaison en utilisant ces compteurs. L'algorithme DSU est organisé en vagues : par exemple, il informe tous les microservices et le *front-end* qu'une nouvelle exécution de l'algorithme DSU commence, il crée les nouveaux microservices et les nouveaux liens entre les microservices pour la prochaine configuration, il attend la validation de la condition de mise à jour avant d'effectuer la reconfiguration, et enfin il informe tous les microservices et le *front-end* que la mise à jour est terminée. Il est important de noter que lors de l'attente de la condition de mise à jour, le rôle de l'algorithme DSU est de faire en sorte que le gestionnaire autonomique reconfigure le système pour que la condition de mise à jour devienne une propriété stable : une fois atteinte, la

condition de mise à jour reste valable jusqu'à l'exécution de la reconfiguration. De plus, nous intégrons les quatre stratégies de mise à jour de la littérature (« versions concurrentes », « messages bloquants », « collaborations bloquantes », et « attente (au plus t secondes) »). Une première version de l'algorithme DSU est mise en œuvre dans le prototype ARBORE.

Cette thèse ouvre plusieurs perspectives.

***Benchmarking* des microservices.** Nous avons réalisé des prototypes pour valider la faisabilité de nos propositions et pour démontrer notre solution. L'exécution de *benchmarks* permettrait de comparer nos contributions avec d'autres solutions du domaine, pour les métriques concernant notamment la complexité des messages, la rapidité de mise à jour (le délai pour atteindre la condition de mise à jour), et la durée de mise à jour (le temps total d'une mise à jour).

Prise en charge d'autres modèles d'interaction asynchrones. Notre travail considère le mode d'interaction client-serveur pour la communication synchrone et le modèle *publier-souscrire* pour les communications asynchrones. Nos modèles peuvent être affinés pour ajouter d'autres modes d'interaction asynchrones comme la communication par flux et les événements complexes.

Évolution au-delà des microservices métier. Dans cette thèse, nous considérons la mise à jour des microservices métier, mais pas la mise à jour des autres parties du système. Par exemple, nous pourrions considérer les changements de version des systèmes de bases de données ou des connecteurs de type *publier-souscrire*. La particularité de ces changements est que les éléments d'infrastructure sont généralement basés sur des plateformes externes ou des technologies de tierces parties.

Stratégies de déploiement. La connaissance des ensembles de remplacement pour exprimer la continuité de service devrait intéresser les administrateurs de systèmes pour permettre des stratégies de déploiement telles que le déploiement *blue/green* ou l'approche *Canari*. Certains outils de gestion existants permettent de gérer automatiquement le déploiement et l'exécution de microservices qui sont enveloppés dans des conteneurs ou d'autres abstractions technologiques spécifiques, par exemple *Kubernetes* ou *Cloud Foundry*. Mais, ces outils ne sont pas conscients des comportements de l'application. Un point de départ serait d'utiliser une approche basée sur une architecture déclarative pour déployer des microservices sur des plateformes PaaS en appliquant différentes stratégies de déploiement.

Tolérance aux fautes. Nos contributions supposent que le système est exempt de fautes. Cela peut être reconsidéré en ajoutant des fautes simples telles que des arrêts francs dans un premier temps. Les défaillances des microservices peuvent, par exemple, être traitées par le gestionnaire autonomique par le biais d'un processus de retour en arrière qui exploite le graphe d'évolution et revient à une branche différente. Pour les défaillances survenant pendant l'exécution de l'algorithme DSU, un retour en arrière vers une vague précédente de l'algorithme DSU pourrait aussi être envisagé.

Contents

Acknowledgements **i**

Abstract **ii**

Content **xiii**

List of Figures **xvi**

Listings **xviii**

1 Introduction **1**

1.1 Research Context 2

1.2 Industry Requirements 4

1.3 Thesis Contributions 6

1.4 Manuscript Organisation 7

2 State of the Art on Dynamic Software Evolution of Microservice Architectures **9**

2.1 Software Architecture of Microservice-based Applications 10

2.1.1 Definitions of software architecture 10

2.1.2 From monolith to modularity 11

2.1.3 From objects to components, services and microservices 12

2.1.4 Microservice architecture 14

2.2 Software Evolution of Software Architectures 16

2.2.1 Basic concepts of software evolution 16

2.2.2 Autonomic computing and model at runtime 18

2.2.3 Role of software architecture in evolution 20

2.2.4 Software evolution for microservice architectures 21
2.3 Dynamic Software Updating 22
2.3.1 Objectives of DSU 23
2.3.2 Characterisation of changes 24
2.3.3 Distributed application model for DSU 25
2.3.4 Consistent update conditions 28
2.3.5 Reaching strategies 32
2.3.6 Distributed termination algorithm 33
2.4 Conclusion 34

3 Runtime Models and Evolution Graph for Version Management of Microservices 36
3.1 Case Study GDE 37
3.2 Runtime Models 38
3.2.1 Model of types 39
3.2.1.1 Configuration types view 39
3.2.1.2 Microservice types view 40
3.2.1.3 Contract types view 42
3.2.2 Model of instances 46
3.3 Evolution Graph 48
3.3.1 Overview of the graph building process 49
3.3.2 Illustrative scenarios 50
3.3.2.1 Patch change 52
3.3.2.2 Minor change 53
3.3.2.3 Major change 54
3.4 Implementation in MIMOSAE 55
3.4.1 PDDL planner 56
3.4.2 Executor 59
3.5 Discussion 60
3.6 Conclusion 62

4 Snapshot-based Dynamic Software Updating of Microservices 64
4.1 DSU Problem 66
4.1.1 GDE use case 66
4.1.2 Role of the DSU algorithm 66
4.1.3 Update conditions 68
4.2 Distributed System and Application Models 69

4.2.1 Distributed system model . . . 71
4.2.2 Distributed application model . . . 72
4.3 Snapshot-Based Update Setting . . . 74
4.3.1 Microservice execution model . . . 74
4.3.1.1 Path of application messages . . . 74
4.3.1.2 Link dependencies . . . 75
4.3.1.3 Message type dependencies . . . 76
4.3.2 Adding continuity of service . . . 80
4.3.3 Essential and non-essential changes . . . 81
4.3.4 Correct dynamic update . . . 83
4.3.5 Snapshot-based definitions of the update conditions . . . 84
4.3.5.1 Quiescence . . . 84
4.3.5.2 Freeness . . . 85
4.3.5.3 Essential freeness . . . 85
4.4 Snapshot-Based DSU Algorithm . . . 86
4.4.1 Termination detection of collaborations . . . 86
4.4.2 DSU algorithm and updating strategies . . . 88
4.4.2.1 Overview of the DSU algorithm . . . 89
4.4.2.2 DSU algorithm for quiescence . . . 89
4.4.2.3 DSU algorithms for essential freeness and freeness . . . 90
4.5 Discussion . . . 92
4.6 Conclusion . . . 94

5 Conclusion and Perspectives 95
5.1 Contribution Summary . . . 95
5.2 Future Work . . . 98

Bibliography 100

Appendix 111

A Implementation Details of Planner and Executor in MIMOSAE 111
A.1 Full PDDL file for MIMOSAE . . . 111
A.2 Architecture of Executor in Autonomic Manager for MIMOSAE . . . 116
A.3 Overview of the proposed Software Evolution Process . . . 117

List of Figures

2.1 MAPE-K control loop 19
2.2 Causal connection between models at runtime and the managed system 20
2.3 Examples of transactions in sequence diagram 26
2.4 Examples of modelling transactions as workflow 27
2.5 State machine diagram of components 28
2.6 State machine diagram with quiescence 29
2.7 State machine diagram of the transition to the tranquillity property . . 30
2.8 Example illustrating several activation periods of the same component . 31

3.1 A global view of GDE illustrative application 38
3.2 Model of types: configuration types view 39
3.3 Model of types: microservice types view 41
3.4 Object diagram for the model of types applied to GDE: a part of microservice types view 43
3.5 Model of types: client-server contract types view 44
3.6 Model of types: publish-subscribe contract types view 45
3.7 Publish-subscribe contract types view: channel-based and topic-based . 47
3.8 Model of instances 48
3.9 Evolution graph of configuration snapshots and configuration types . . 49
3.10 Example of object diagram of type model with a possible minor change 53
3.11 Example of object diagram of type model with a possible major change 55
3.12 Scenario implemented in MIMOSAE 58

4.1 Sequence diagram of the use case `AttachFileToProject`—publish-subscribe communication 67
4.2 Global view of DSU algorithm for microservices 68

4.3 Sequence diagram of use case `AttachFileToProject`—client-server communication . . . 70
4.4 Multigraph of message type dependencies for the use case scenario `AttachFileToProject` . . . 79

A.1 Communication diagram of executor's general functioning . . . 116
A.2 Overview of software evolution process . . . 118

Listings

3.1 Example of the PDDL domain file for MIMOSAE 57
3.2 Example of the PDDL problem file for the scenario of minor version change 59
3.3 Example of the plan generated by the PDDL planner 60
A.1 PDDL domain file for MIMOSAE 111
A.2 PDDL problem file for creating GDE architecture with synchronous client-server calls 113
A.3 Configuration plan generated for creating GDE architecture with synchronous client-server calls 113
A.4 PDDL problem file for a minor revision with addition of logging in the GDE architecture 114
A.5 Configuration plan generated for the corresponding minor revision of the GDE architecture 115

Chapter 1

Introduction

Contents

1.1 Research Context 2

1.2 Industry Requirements 4

1.3 Thesis Contributions 6

1.4 Manuscript Organisation 7

Microservice architectures contribute to building complex distributed systems as sets of independent microservices. The modularity of distributed microservices facilitates their independent replacement and upgradeability, enabling agile DevOps and continuous evolution. Modelling and managing the rapid evolutionary changes of the microservice-based applications is the basis of solutions to perform dynamic updates of microservices with minimal halting time. This thesis aims at, on the one hand, helping engineers to perform modelling and version management activities for microservices, and on the other hand, improving continuity of service during dynamic updating of microservice-based applications.

We begin this introduction chapter with the research context of this thesis (Section 1.1). We establish the scope of our work and motivate the interest of the problems that we address in this thesis. Because this thesis is supported by "Conventions industrielles de formation par la recherche (Cifre)" that is a cooperation between a company and a research laboratory, with the help of our our industry partner, Electricité de France (EDF), we identify industrial requirements and provide them in Section 1.2. We then summarise the contributions of our work in Section 1.3. Finally, we conclude this chapter with the presentation of the structure of this manuscript (Section 1.4).

1.1 Research Context

Many attempts and efforts have been made to respond to the growing scale and evolution of software architectures of enterprise applications. One of the approaches is modularity. During the last two decades, the software architecture domain has gone through the process of transforming monolithic architectures into distributed component-based architectures and service-oriented architectures, and more recently, into microservice-based architectures. In recent years, microservices have become a popular software architecture paradigm and have been successfully adopted by many prominent large companies to achieve high agility, maintainability and sustainability of their software systems. In this approach, each microservice provides one (small) functionality with one (small) business objective, and can be leveraged by one or more other microservices through lightweight communication [Newman, 2015]. Microservices are built using different technology stacks and are managed by different teams. The two commonly used protocols are HTTP request-response with resource APIs and lightweight messaging [Lewis and Fowler, 2014]. With their modular design, microservices act as independent units of development, deployment, evolution, and runtime control.

In practice, while microservice architectures have brought several benefits, the adoption path for microservices is not easy [Killalea, 2016]. Microservices need to evolve continuously in response to various changes such as changes in human requirements, technology, and the application environment. The maintenance and evolution costs of these distributed systems increase rapidly with the number of microservices. Furthermore, since the emergence of agile development [Ebert et al., 2016] and continuous practices (continuous integration, delivery, and deployment) [Humble and Farley, 2010], there is a trend towards even more frequent and rapid evolutionary changes of microservice-based applications. As each microservice has its own life cycle, not all microservices in an application evolve at the same pace. They can evolve independently and frequently to introduce new features, remove existing obsolete features, change the implementation of a given microservice to correct performance or security problems, etc. Changes to microservices cannot be foreseen at the time of system creation, and may occur when the microservices are active providing round-the-clock services.

Changes in a distributed system are usually specified declaratively in terms of system structure. The specification of change is derived by comparing the desired configuration with that of the current system [Kramer and Magee, 1990]. Similarly, microservice-based applications can specify their changes in terms of their architectures, which makes it possible to mirror changes as dynamic translations between architectural

models and configurations. Applying one or more changes to microservice architectures is performed by an evolution process of moving from the current application version to a new version. Evolving a microservice means, from a development view, bringing into play a new version of its artefacts, and from a configuration view, deploying new instances, linking these new instances to other microservices, unlinking obsolete instances before removing them, etc.

The changes applied to microservices make the system more complex over time so that it is impossible for any one person to memorize every change. Also, microservices in one application can be managed by different autonomous teams. Microservices and their dependencies may not be controlled by only one team. Coordinating and ensuring alignment among different teams becomes difficult. Traceability of evolutionary changes enables to track changes in microservices across the lifecycle of development and maintenance, that is, to track what is currently happening and what has already been changed. It helps engineers understand how the system evolves, helps them choose the appropriate version they want to use, and helps them source the reason of changes when comparing two versions.

In addition to specifying and performing changes by modelling microservice architectures and their version management, it is also necessary to provide approaches for managing and controlling the evolutionary changes in the running microservice-based applications. A traditional way to apply changes to running software systems is stop-replace-restart, which is also called offline update. However, this offline approach is not suitable for updating microservice-based applications, especially not for their long-running or frequently-executed activities, because it can result in long service interruptions that are unacceptable for customers. Thus, online updating is essential to improve the continuity of services. Updating is said to be online or dynamic if the updating process is carried out at runtime to adapt to changes in the execution environment without stopping the execution of the whole system. This is called dynamic software updating.

Dynamic software updating should not only guarantee that the resulting application is correct, but it should also ensure that the ongoing activities are properly completed before the system is configured, i.e. the involved stateful microservices should not be active. Otherwise, the evolution would be considered unsafe and inconsistent. Since the seminal work of [Kramer and Magee, 1990] who proposed an update condition to maintain safe updating in component-based systems with synchronous communication, this problem has received regular attention with contributions from time to time: the works of [Vandewoude et al., 2007, Ma et al., 2011, Baresi et al., 2017] are contributions to dynamic updating for synchronous distributed transactions in component-based

systems; the very recent work of [Sokolowski et al., 2022] extended previous solutions to asynchronous workflows. We want to mention that our contribution to dynamic software updating with client-server and publish-subscribe interactions between microservices has been prepared in parallel to this very recent publication.

The objective of this thesis is to address the following two issues: (i) How to build a unified and efficient version management for microservices and how to specify and realise the traceability of changes in microservice-based applications? (ii) When can microservice-based applications, especially those with long-running activities, be dynamically updated and how should the safe updating be performed to ensure service continuity?

1.2 Industry Requirements

Microservice architectures have now been popularly applied in software companies, such as Netflix, Amazon, etc., in regard to our industrial partner, Electricité de France (EDF), an electric utility company whose primary business is electricity production and supply, rather than the commercialisation of software products. An important requirement for these non-software-oriented companies is to rationalise the cost of the development and maintenance of their information systems, which are the supporting tools to increase the productivity of their primary business. They promote the reuse of existing heterogeneous software solutions that can be organised as microservices. These microservices are said heterogeneous because they are developed and maintained by different development teams and operation teams, which may be internal teams or subcontractors, and because these teams may have different technology preferences. In this situation, even simple changes in one microservice can lead to cross-team discussions that require the sharing of software architecture artefacts that trace software evolution. Microservice providers often deploy multiple versions in parallel, offering some specific versions to certain customers or previous versions for legacy systems. In this case, the updates of these heterogeneous microservices may go through organisational boundaries, and add dynamics and complexity to software evolution.

There exist several frameworks or tools widely used for microservices in practice. Even though each of them may focus on one aspect or may not be designed and developed specifically for microservices, they can be of great help and be composed to work together. We consider some categories of industry frameworks or tools that are able to support evolutionary activities in one way or another: (i) container frameworks for packaging and running microservices (e.g. Docker [1]), (ii) container orchestration tools

1. Docker: https://www.docker.com/

for managing the life cycle of containers (e.g. Kubernetes [2]), (iii) service mesh platforms for governing service-to-service communication and message exchange through a sidecar proxy added next to each microservice (e.g. Istio [3]), (iv) cloud platforms for enabling on-demand assembly of various predefined management solutions (e.g. Amazon Web Services [4]), and (v) automation tools for facilitating continuous integration, delivery, and deployment (CI/CD) when releasing new versions of microservices (e.g. Jenkins Pipeline [5]).

These tools automate the process of getting microservices releases from a version control system. They also realise automated roll-outs and rollbacks of containerised microservices depending on the desired state in configuration files through some predefined deployment strategies, such as rolling update [Alex, 2022], canary release [Sato, 2014], etc. Therefore, these tools simplify the deployment and production release steps in the process of carrying out microservice updates. But before these steps can take place, engineers still have many tasks to handle. Note that, in this thesis, we limit ourselves to the evolution of business microservices themselves, but not focus on changes in other parts, such as infrastructure, etc.

In collaboration with our industrial partner, we have identified several evolution-related pending requirements:

- Due to the heterogeneity of microservices in the ecosystem of the company, a global and unified representation of microservice architectures should be provided to architects, developers, and administrators.
- Software evolution for microservice architectures should be modelled and follow well-defined rules when a microservice changes, or some microservice dependencies change.
- The system should be able to handle the deprecated versions of microservices in order to free up unnecessary resources.
- The system should allow the traceability of the evolutionary changes to the system, and make it possible to switch to a specific version or to a previous version.
- The system should provide a continuous service and minimise the duration of client interruptions as much as possible. It should also not block parts of the system not affected by reconfigurations of a given set of microservices.

2. Kubernetes: https://kubernetes.io/
3. Istio: https://istio.io/
4. AWS: https://aws.amazon.com/
5. Jenkins: https://www.jenkins.io/

1.3 Thesis Contributions

Based on the research context and industry requirements mentioned in the previous sections, we work on the evolution of microservice-based applications and our main contributions are the following ones:

1. **Runtime models and evolution graph for modelling and tracing version management of microservices.** This contribution is awaited by engineers for governing heterogeneous microservice evolution activities in a unified manner, such as version management and reconfiguration deployments. We first propose runtime models to describe the software architecture of evolving microservice-based application. The software architecture is represented by two models: the microservice type model for abstracting the structure of instantiable elements and the microservice instance model for representing replicas of the corresponding deployable elements. These two models are built by engineers at design time and then used at runtime. Each element in the type model is syntactically assigned a version number, and each element of the instance model conforms to a type element and is also syntactically assigned a version number. In addition, our models also distinguish synchronous (RPC calls) from asynchronous (publish-subscribe) interactions within information systems. We then build up an evolution graph for tracing evolution histories at the granularity of a configuration, which includes a set of microservices, not just a single microservice. We record the trajectory of how a microservice-based application evolves over time as changes are applied. This evolution graph is composed similarly by two cross-linked sub-graphs of configuration type snapshots and configuration instance snapshots. These snapshots are photos “à la Git” of the evolution of the microservice architecture.

2. **Snapshot-based dynamic software updating of microservices.** The first contribution provides a knowledge base to be manipulated by an autonomic manager middleware service in various evolution activities for microservice-based applications. Some additional efforts are required for safely and consistently updating microservices at runtime, which is the focus of this second contribution. During the evolution process of a running application evolving incrementally from configuration to configuration, we propose a novel approach to determine the answers to the two following questions: (1) When can updating be performed safely (i.e. the update condition)? (2) How can the impacted microservices reach and maintain a consistent state (i.e. updating strategies)? The novelty is in answering these questions by constructing consistent distributed snapshots of the

microservice-based application. These snapshots are photos, i.e. global states, of the execution of the distributed application; they must be consistent to be meaningful. We use these consistent distributed snapshots for evaluating the update conditions and realising the update strategies. In addition, since dynamic software updating is expressed in terms of a configuration rather than of a single component, we define continuity of service.

Two articles were published during this thesis and another article is in preparation.

— [Wang, 2019]: "*Towards service discovery and autonomic version management in self-healing microservices architecture*", Doctoral Symposium, in Proceedings of the 13th European Conference on Software Architecture, Volume 2, pages 63–66, Paris, France.

— [Wang et al., 2021]: Y. Wang, D. Conan, S. Chabridon, K. Bojnourdi, and J. Ma, "*Runtime models and evolution graphs for the version management of microservice architectures*", in Proceeding of the 28th IEEE Asia-Pacific Software Engineering Conference, pages 536–541, Taipei, Taiwan.

— (In preparation): Y. Wang, D. Conan, S. Chabridon, K. Bojnourdi, and J. Ma, "*Snapshot-based Dynamic Software Updating for Microservice-based Applications*"

Also, an open source prototype has been implemented to validate the first contribution: MIMOSAE (for "MIcroservices MOdel for verSion mAnagement with Evolution graph"), `https://gitlabev.imtbs-tsp.eu/mimosae/mimosae`. The implementation of a second prototype is in progress for demonstrating the second contribution: ARBORE (for "ARchitecting Based on microservice versiOns with REconfiguration"), and will be soon available at the following URL: `https://gitlabev.imtbs-tsp.eu/mimosae`.

1.4 Manuscript Organisation

This manuscript is organised into four chapters.

In Chapter 2, we present the background knowledge and the state of the art of the research fields addressed in this thesis. We position our work in the context of software evolution, and we then go over the state of the art in software evolution of microservice architectures and dynamic updating of microservice architectures.

In Chapter 3, we present the first contribution of this thesis: runtime models and evolution graph for version management of microservices. It focuses on coping with the requirements of modelling and tracking the evolution of heterogeneous microservices. We begin with an illustrative microservice-based application from an industry project

that involves different technology stacks and interaction modes. We then explain our proposed runtime model that can be used to specify structural abstraction and deployment configuration of microservice architectures, and an evolution graph that is built to trace model element versions (microservices, etc.). Then, we implement our approach with the help of an AI Planner in order to automatically generate a reconfiguration plan, which is executed using Kubernetes and Docker APIs.

In Chapter 4, we present the second contribution of this thesis: a snapshot-based approach for dynamically updating microservice architectures. It addresses the issue of dynamic software updating (DSU) that guasantee both consistency during updating and continuity of service. We complement our microservice architecture models of Chapter 3 in order to formulate properties such as continuity of service and essential change. Then, we use the concept of consistent distributed snapshot of distributed systems to express properties such as update conditions and update correctness. Furthermore, our DSU algorithm is also based on consistent distributed snapshots to look for and maintain safe update conditions with different update strategies.

In Chapter 5, we conclude this thesis and discuss several perspectives as future works.

Chapter 2

State of the Art on Dynamic Software Evolution of Microservice Architectures

Contents

2.1 Software Architecture of Microservice-based Applications 10
2.1.1 Definitions of software architecture 10
2.1.2 From monolith to modularity 11
2.1.3 From objects to components, services and microservices 12
2.1.4 Microservice architecture 14
2.2 Software Evolution of Software Architectures 16
2.2.1 Basic concepts of software evolution 16
2.2.2 Autonomic computing and model at runtime 18
2.2.3 Role of software architecture in evolution 20
2.2.4 Software evolution for microservice architectures 21
2.3 Dynamic Software Updating 22
2.3.1 Objectives of DSU 23
2.3.2 Characterisation of changes 24
2.3.3 Distributed application model for DSU 25
2.3.4 Consistent update conditions 28
2.3.5 Reaching strategies 32
2.3.6 Distributed termination algorithm 33
2.4 Conclusion 34

This chapter presents the primary concepts, principles and solutions related to our work. We position this thesis in the fields of the modelling and dynamic evolution of microservice architectures.

In Section 2.1, we start by defining the concept of software architecture with the mutation from monolithic architectures to different granularities of modularity: from objects to components and services, and now to microservices. We then detail the definitions and the characteristics of microservice-based architectures, as well as the benefits and challenges of interest for our work.

In Section 2.2, we introduce software evolution by focusing on runtime software evolution. We argue that software architecture can provide a foundation and play an important role for systematic runtime software evolution.

In Section 2.3, we describe the dynamic software updating problem. Some interesting requirements and goals of this problem are firstly introduced with the corresponding terminology. Next, we present the state of the art on the dynamic update conditions to be reached before updating may take place, the updating strategies, and the basic algorithm of distributed termination detection.

In Section 2.4, we conclude by highlighting the assumptions and the choices made by the approaches covered in this chapter.

2.1 Software Architecture of Microservice-based Applications

In Sections 2.1.1–2.1.3, we describe how the field of software architecture has developed and evolved, up to the emergence and popularity of microservice architectures. In Section 2.1.4, we then introduce the fundamental concepts of a microservice architecture, including its definitions, characteristics, benefits, and weaknesses.

2.1.1 Definitions of software architecture

As one of the fundamental disciplines in software engineering today, software architecture gives a blueprint for complex software systems. It provides a high-level abstraction of the overall structure, behaviour, and properties of software systems. Software architecture typically plays an important role as a bridge between the requirements and the implementation [Garlan, 2000]. It supports early design decisions for teams to satisfy all the technical and operational requirements, and to improve essential software system qualities. It also helps stakeholders better understand and analyse how software systems are designed, implemented and deployed, whether building a new software system, evolving a current software system, or modernising a legacy software system.

The origin of the concept of software architecture can be traced back to the 1969 NATO Conference on Software Engineering Techniques (the report of this meeting can be found in [Randell and Buxton, 1970]). Some of the most prestigious pioneers in the field, such as Edsger Dijkstra, Per Brinch Hansen, Friedrich Bauer, were present at this conference and emphasized the need to make the subject of software architecture public and focused. Subsequently, software architecture as a distinct discipline has emerged and flourished in both industry and academia in the 1990s.

A seminal work published in [Perry and Wolf, 1992] has formulated a typical model of software architecture consisting of architectural elements, their form and rationale, as a triple: *"Software Architecture = {Elements, Form, Rationale}"*. The *Elements* are distinguished into the three classes of data, processing, and connection. The *Form* highlights the constraints on these elements, including weighted relationships and properties of the elements. And the *Rationale* explains the motivation for selecting elements and their form, and satisfying the constraints ranging from basic functional requirements to extra-functional requirements.

In this thesis, the definition we choose is the one provided in [Bass et al., 2003]: "*The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both* ". The software elements are for instance components, services, and microservices; the relations among them are realized into connectors; and the properties configure both software elements and connectors [Medvidovic and Taylor, 2010].

Meanwhile, the industry practitioners have also started to leverage architectural design in the development of their products. For example, the work of [Soni et al., 1995] discussed software architecture in industrial applications and separated different engineering concerns into four views, while Kruchten proposed a model with five concurrent views in the so-called *"4+1 View Model"* [Kruchten, 1995].

With the development and maturity of the discipline, several reusable and proven good solutions are usually codified into architectural styles and patterns to address recurring problems [Medvidovic and Taylor, 2010]. Some well-recognized examples, which we talk about in the following sections, are monolithic architecture, component-based software engineering, service-oriented architecture. Of course, the microservice architecture we are targeting in this thesis is also one of them.

2.1.2 From monolith to modularity

The software applications that are designed in a single tier where all the elements run in a process decomposed into threads that share the resources (memory, databases,

files, etc.) are called monoliths. This monolithic style is considered as a standard way to start developing an application. It facilitates application development, deployment, and updating as long as the size of the codebase is relatively small. However, without modularity, the rapidly increasing software complexity leads to *monster monoliths* becoming unmaintainable systems and caused a *software crisis* [Naur and Randell, 1969]. Any change in one element of a monolith may require redeploying and rebooting the whole application. To resolve the side effects of a monolith, software system development has then followed the way to modularity and shifted the emphasis from traditional monoliths to distributed systems. The modularity of software architecture describes the degree to which architectural elements can be separated and recombined [Schilling, 2000]. One of its objectives is to create and maintain independently executable modules to maximize their code reuse.

2.1.3 From objects to components, services and microservices

The applications in object-oriented design are often modelled as complex hierarchies and graphs of classes. The way to achieve reusability is to inherit code from an existing base class and specialize its behaviour. This approach is typically referred to as white-box reuse [Ravichandran and Rothenberger, 2003]. But once the classes are compiled, the result is still a large monolithic binary code. Also, it is necessary to be familiar with the details of the base class implementation when subclassing.

When the focus has shifted from the relationships between classes in source code to independent and interchangeable binary building modules, component-based architectures have emerged. We choose the well-known definition given by [Szyperski et al., 2002] for this architectural style: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*"

The approach is based on black-box reuse, i.e. the implementation of each component is entirely hidden behind interfaces that explicitly specify the provided functionalities and context dependencies. The principle is then to build distributed applications as functional building blocks (components) interacting with each other via the provided and required interfaces. The software infrastructure needed to connect components and to handle common requirements can be supported by some component technologies platform or framework. In other words, components typically rely on platform-specific development and runtime technologies [Erl, 2008]. For example, components can be built using Distributed Component Object Model (COM/DCOM) [Thompson

et al., 1997], Java Platform Enterprise Edition (Java EE) with Enterprise Java Bean (EJB) [Oracle, 2022], Common Object Request Broker Architecture (CORBA) [OMG, 2022], .NET [Microsoft, 2022].

The combination of object-oriented programming (OOP) and component concepts gave birth to Service Oriented Architectures (SOA) [Dragoni et al., 2017]. A *service* is another granularity for achieving modularity besides object and component. In [Erl, 2016], the authors give the following definition of this style of architecture: "*Service-oriented architecture (SOA) encourages individual units of logic to exist autonomously yet not isolated from each other. Units of logic are still required to conform to a set of principles that allow them to evolve independently, while still maintaining a sufficient amount of commonality and standardization. Within SOA, these units of logic are known as services.*"

It results that a service can be considered as an autonomous and independent computational entity with well-known interfaces that logically realize a business activity or a supporting task. Compared with components, which emphasize composition and reusability, services highlight business abstraction and interactivity. Furthermore, services are independent of vendors, technologies and platforms, whereas it is not always the case for components. As a consequence, SOA is viewed as a framework or a design pattern where enterprises build, deploy, and manage these services [de Giacomo et al., 2021]. The conceptual model of SOA involves the three roles of service provider, service consumer, and service repository. One of the common practices to implement SOA is with Web services [Alonso et al., 2004], where services are defined by standardized service contracts using for instance the Web Service Description Language (WSDL), communicate through SOAP messaging protocol, and are discovered by a UDDI (Universal Description Discovery & Integration) registry. The services interact with each other to provide a more complex business process, also called a workflow. Specific workflow languages allow to model and orchestrate the behaviour of these workflows, one of the most well-known being the Web Services Business Process Execution Language (WS-BPEL) [OAS, 2007]. In practice, an integration backbone called Enterprise Service Bus (ESB) is often used as a middleware to provide technical infrastructures and tools to facilitate hosting and implementing services for SOA, including message routing, service orchestration, monitoring, versioning, and security [Chappell, 2004].

Because the focus shifts from encapsulation and componentization to independent development and deployment, the term "microservice" was first introduced as a guideline of a software architectural pattern in 2011 at a workshop of software architects [Dragoni et al., 2017]. While there is some debate on whether microservices are just a variant of SOA, it is safe to say that they are related because microservice architecture is

a further iteration of SOA, but that does not mean that they are the same. The service granularity between the two architectures is significantly different. The size of SOA services ranges from fine-grained application services to coarse-grained enterprise services. The prefix "*micro-*" indicates that each service is fine-grained and limited to a single business function, following the Unix philosophy of "*doing only one thing and doing it well*" [Raymond, 2003]. Concerning sharing, for certain technical services such as databases and logging, SOA tries to maximise sharing with enterprise-level services and databases, while microservice architecture intends to minimize this sharing with the concept of "*bounded context*": each microservice typically has its own database, logging system or other purely technical functions. Furthermore, SOA and microservices rely on different technology stacks [Jamshidi et al., 2018]. Typically, SOA uses heavyweight technologies such as SOAP, WSDL, ESB, whereas microservices tend to use lightweight technologies such as REST, publish-subscribe broker, container.

2.1.4 Microservice architecture

Software architectures of enterprise applications are getting larger and more complex, and many attempts and efforts have been made to respond to the growing scale of software and increasing network requests. Microservice architecture is one of the architectural pattern that makes it possible to design, develop and deploy scalable and flexible systems. Now that it is clear where microservices come from along the way of software architecture and why they came out, we present their principles and how they can be used properly according to their advantages and challenges.

Presently, there is no agreed precise definition of a microservice architecture. We list some of the interesting existing definitions in the following. A popular definition was early described in [Lewis and Fowler, 2014] for microservice architectures: "*The microservice architectural style is an approach to developing a single application as a suite of 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.*"

This definition includes several characteristics that architects expect microservice architectures to exhibit. Some other definitions of microservices also exhibit one or more characteristics, such as:

— "*Microservices are small, autonomous services that work together*" [Newman, 2015].

— "*A microservice is a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility*" [Thönes, 2015].

— "*A microservice is a cohesive, independent process interacting via messages. The term 'cohesive' indicates that a service implements only functionalities strongly related to the concern that it is meant to model. A microservice architecture is a distributed application where all its modules are microservices*" [Dragoni et al., 2017].

Any architectural style requires trade-offs. Microservices provide benefits, but also come with costs. It is a good practice to evaluate the strengths and weaknesses in context before using it. We now list some common characteristics of microservice architectures seen in the definitions.

Modular with independent services. Microservice architectures use services as the unit of modularity, which has explicit offered and required interfaces and handles a single responsibility. When modelled as services, their capabilities are exposed to other collaborators by using remote call mechanisms over the network to enforce the separation. Each service is a unit of deployment, and thus of scalability. Changes to any single service are expected to only result in redeployment of that service or as few services as possible because of its loose coupling. Even if changes in microservices and their interfaces affect some coordination, the cohesive service boundaries and additional evolution mechanisms in the service contracts could help minimize the impact. For example, in practice, engineers can deploy updated versions of microservices to production in a shorter time of rapidly changing business environments by using modern container tools or other DevOps practices [Waseem et al., 2020].

Around Business Capabilities. As formulated in the *Conway's Law*, working on big problems with distributed cross-team and large codebases does not improve productivity: "*Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure*" [Conway, 1968]. Thus, microservices organised around business capabilities may help to better align the architecture with the organization and try to achieve the best balance of team size and productivity. The size of a microservice should be "*small enough and no smaller*" [Newman, 2015] to balance the benefits of small size with the resulting

complexity. A famous example is the view of Amazon's *two pizza* teams that a team should be fed with two pizzas [Chapman, 2014]. In addition, small teams organised according to microservices need to be cross-functional to manage the whole lifecycle of the services they are developing.

Lightweight communication. Two of the most common protocols used in microservice architectures are HTTP request-response with resource APIs (e.g., HTTP REST) and messaging via a message broker (e.g., AMQP [AMQP Consortium, 2010], Apache Kafka [1], and MQTT [OASIS, 2019]). Then, the two ways to build collaborations are orchestration and choreography [Peltz, 2003]. Orchestration requires a central service that sends requests to services and oversees the business process by receiving responses. Choreography considers no centralisation, uses events and the publish/subscribe paradigm to establish collaboration, and track message sequences. The culture of microservices tends to minimize centralisation and makes them more inclined to use choreography.

Technology heterogeneity. Theoretically, each microservice has the freedom to select a different set of technologies for programming languages, libraries, data stores, etc. As a consequence, the initial technology choices should not severely limit the ability to use new languages and frameworks in the future. Although microservices allow teams to more easily embrace with new tools, too much is as bad as too little, most organisations encourage the use of a limited set of technologies.

2.2 Software Evolution of Software Architectures

In this section, we introduce the research area of software evolution (Section 2.2.1). We notice the changes that may occur at runtime, and investigate methods to automate the system management and adaptation at runtime (Section 2.2.2). Furthermore, we also highlight the role of software architecture in software evolution (Section 2.2.3), and present some current works on software evolution for microservice architectures (Section 2.2.4).

2.2.1 Basic concepts of software evolution

In software engineering, software evolution is referred to as the process of developing, maintaining, and updating software in its entire life cycle. Software changes are

1. Apache Kafka: https://kafka.apache.org/

inevitable in software development in order to respond to evolving requirements, performance and environmental pressures [Godfrey and German, 2008].

As a pioneer in the field of software evolution in the last century, Lehman and his collaborators have formulated a set of observations about software evolution, called the *Lehman's Laws of Software Evolution* [Lehman, 1980, Lehman et al., 1997, Lehman and Ramil, 2002]. Even though the software research area has changed a lot since then, some of their rules, especially those on software size and complexity, still hold true: for example the law observing that the size and complexity of a software system will continue to increase over its lifecycle. Lehman's work mainly focused on the nature of software evolution and the properties of evolutionary phenomena for monolithic systems produced by only one team. However, as the use of components, services, and microservices becomes more common, researchers began to reconsider and continually adapt Lehman's Laws to modern software [Godfrey and German, 2014], including in the context of open source projects [Yu and Mishra, 2013] and agile development [Sindhgatta et al., 2010].

The work of [Mens et al., 2003] proposed a taxonomy of software evolution based on the characteristics of software changes. In their taxonomy, the time at which the change occurs greatly affects the approaches to software evolution:

— Compile-time evolution, also called *static evolution* [Cook et al., 2001], concerns primarily the source code of the system: e.g. code modularization support for maintainability [Parnas, 1971], code refactoring [Fowler, 2018]. Software in this case needs to be recompiled for the changes in a new executable system.

— Load-time evolution considers the changes that occur when software elements are loaded into an executable system: e.g. modifications of binary JAVA class files during load-time [Chiba, 2000, Kniesel et al., 2001].

— Run-time evolution, also called *dynamic evolution* [Cook et al., 2001], allows software to evolve dynamically by hot-swapping components and integrating new components without stopping the execution of the distributed application.

The first two are not what we are going to discuss in this thesis, but we focus on run-time evolution that allows the software system to adapt to changes at runtime. We explore some issues of runtime evolution such as how these runtime changes are specified and managed, and how to ensure consistency, correctness, or other desired properties of these runtime changes.

2.2.2 Autonomic computing and model at runtime

The kinds of systems that can benefit from runtime software evolution include, but are not limited to, long-time running systems that need online modification and adaptive systems that need to adapt to changes in the execution environment [Oreizy et al., 1998, Dowling and Cahill, 2001]. One of the trends of interest in the research community is about how to manage and control the evolutionary activity of these systems with minimal human intervention and less downtime. So, we introduce the MAPE-K approach to automation and the model-at-runtime approach to reflect the architecture of the distribution application to evolve.

Automation of evolution tasks. In fostering automation of the evolution tasks, the concept of "*autonomic computing*" can be considered to enable software systems to manage themselves to a certain degree based on high-level objectives. Among the self-managing characteristics, also called self-abilities, self-management includes activities such as configuration, optimisation, healing or protection that require no or minimal human intervention by using some autonomic control mechanisms [Kephart and Chess, 2003, Huebscher and McCann, 2008]. It makes it possible to free system administrators and other stakeholders from maintaining long-time running systems and frequent adaptive systems.

The structure of the self-adaptive process includes a control loop. A generic model of the autonomic control loop is known as the MAPE-K control loop [Computing et al., 2006]. Figure 2.1 presents the structure of an autonomic element that consists of managed elements and an autonomic manager that controls the managed elements. In our case, the managed elements are for instance microservices, connectors, and database systems. The autonomic manager organises the control process around four activities "Monitor", "Analyse", "Plan", and "Execute" that share a "Knowledge base". Note that this centralised autonomic manager can be distributed by creating several instances. For the sake of simplicity, we limit ourselves to one single autonomic manager in this thesis.

The autonomic manager is in charge of the following activities:

- Monitor: The monitoring activity collects data from the managed elements and its environment, and then aggregates the collected data according to the content of the Knowledge element, which can be later manipulated by other activities.
- Analyse: The analysis activity determines decisions about the need for adaptation and its goals by using information from the monitoring activity and from the knowledge base.

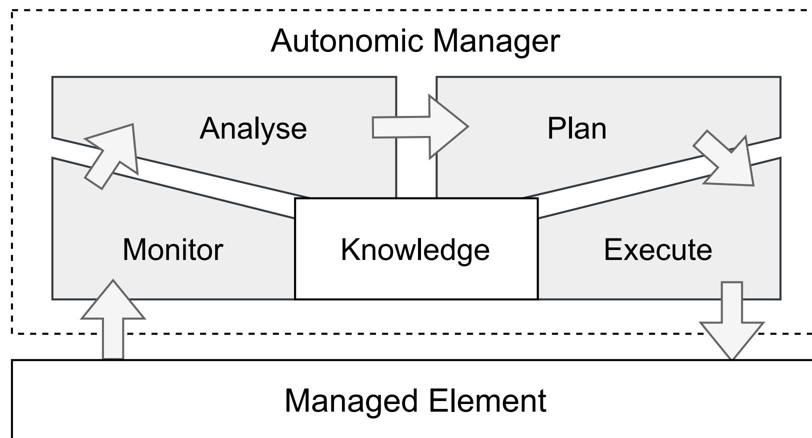


Figure 2.1 – MAPE-K control loop (adapted from [Kephart and Chess, 2003])

— Plan: The planning activity receives the adaptation decisions and builds a procedure consisting of one or more necessary configuration or deployment actions to achieve the adaptation goals.

— Execute: The execution activity applies the adaptation plan and applies changes to the managed system at appropriate moments.

— Knowledge base: This is an element shared between the four activities, and it maintains rules, properties, models and other kinds of data.

Three levels of adaptation can be distinguished. In *manual adaptation*, engineers manually adapt the system, e.g. the work of [Buisson et al., 2016]. In *automatic adaptation*, the system is a self-adaptive system, e.g. the work of [Florio and Di Nitto, 2016, Banijamali et al., 2020]. In *semi-automatic adaptation*, which lies between these two extremes, one part of the control process is realized by engineers and another part by the autonomic manager, e.g. in [Phung-Khac, 2010, Huynh, 2017], engineers decide an adaptation target and an autonomic manager plans and then executes the adaptation process. Our work is applicable to semi-automatic adaptation.

Model at runtime. A runtime model provides a reduced representation of the heterogeneous software elements that are available in the implementation repository and of the managed elements of the running application. It plays a role as a live element to reflect the state of the system and its execution. As defined in [Blair et al., 2009], "*a model@run.time is a causally connected self-representation of the associated system that emphasizes the structure, behaviour, or goals of the system from a problem space perspective.*"

Using computational reflection [Maes, 1987], a runtime model is causally connected with the underlying executing system, and every change in the runtime model will correspondingly lead to a change in the structure or behaviour of the system, and *vice versa* [Bencomo et al., 2019]. Figure 2.2 shows the causal connection between runtime models and the system. Software adaptations are applied to models maintained at runtime before being executed in the managed applications. The high-level abstraction of heterogeneous application through runtime models provides a unified view for the autonomic manager, and helps the autonomic manager to build the knowledge base shared by the control loop.

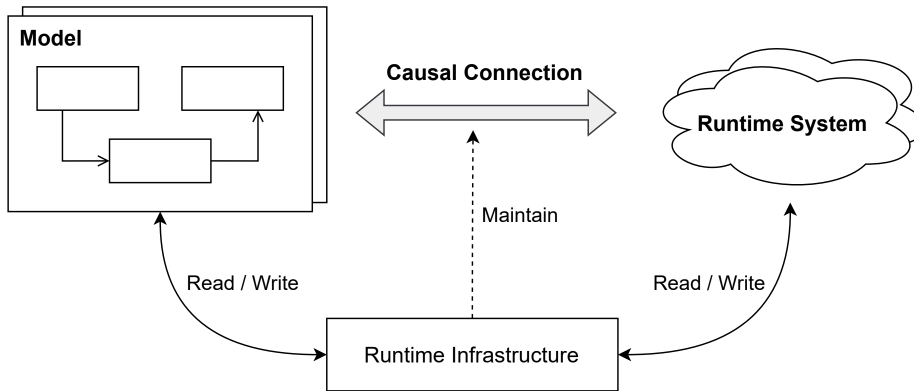


Figure 2.2 – Causal connection between models at runtime and the managed system (adapted from [Oreizy et al., 1998, Huang et al., 2009])

2.2.3 Role of software architecture in evolution

One of the goals of the software evolution research is to design and build models that can be used to describe the evolution of software systems from the past to the present and into the future, especially for runtime software evolution. Ideally, such models can address both qualitative and quantitative properties of systems, as well as some predictive power both in the short and long terms [Godfrey and German, 2008].

As presented in Section 2.1, a software architecture models the structure and the behaviour of a system, including the software entities and the relationships between them. Architectures shift developer focus away from source codes to high-level architectural abstractions. The authors of [Oreizy et al., 1998] have highlighted the need for a systematic approach to support runtime changes, and argued that software architecture can provide the basis for systematically handling runtime changes. Also, dynamic software architectures can be used to build dynamically evolvable software systems by supporting the self-management of systems at runtime and acting as the knowledge base in the control loop [Dowling and Cahill, 2001].

Dynamic software architectures used for runtime software evolution can be modelled by using several approaches, such as formal mathematical methods (e.g. [Wermelinger and Fiadeiro, 2002, Grunske, 2005]), Architecture Description Language (ADL) specification (e.g. [Barais et al., 2008, Garlan et al., 2009]), model at runtime (e.g. [Morin et al., 2009, Vogel and Giese, 2010]). In our work, we bring to play the model-at-runtime approach.

2.2.4 Software evolution for microservice architectures

Microservice architectures emphasize evolvability [Bogner et al., 2019], because the modularity of microservices provides capability to evolve independently and makes microservices easier to change and replace. Also, *evolutionary design* is proposed by [Lewis and Fowler, 2014] as one of the main expected microservice characteristics. These properties provide a beneficial theoretical basis for software evolution. In recent years, the area of software evolution has offered concrete and universally applicable solutions for microservices to support runtime evolution.

Examples that have been published so far include, but are not limited to, the following studies.

The authors of [Sampaio et al., 2017, Sampaio et al., 2019] proposed a service evolution modelling approach that combines static and dynamic information to generate at runtime a representation of the evolving microservice-based system. The objective is to trace software architecture evolution of running applications by monitoring the system.

The authors of [Esparrachiari et al., 2018] discussed several strategies for tracking and controlling microservices dependencies, and argued that tracking dependencies is a passive approach and controlling dependencies is an active one.

The authors of [Tao, 2019, Boyer et al., 2018] proposed a declarative architecture-based approach for microservices, called `DMU`. DevOps teams specify the desired target architecture of the application and choose the strategy for reaching the target architecture. The target architecture defines how microservices instances are configured over the PaaS sites. The strategy defines how to reach the target architecture. A step-by-step transition allows reaching the desired target architecture through a deployment strategy. In case of failure when taking a step, DevOps teams can change some configuration in the target architecture, rollback to the initial architecture, or roll forward to a new target architecture.

The authors of [Ma et al., 2019] presented a tool for monitoring microservice-based systems and generating visualized version-based service dependency graphs through

source code analysis. The dependency graph is computed through a series of chain searches, and version management is based on chain manipulations.

The author of [Akbulut and Perros, 2019] focused on the API versioning of microservices in their URI or HTTP headers, and extends the API gateway design pattern to orchestrate the requests for different versions of APIs with a view to managing the virtual hardware configuration of containers. They modified the Gateway entity by installing several functionalities, such as intelligent routing, observing other ecosystem entities, and scale up or down services based on fuzzy logic.

In the work of [Liu et al., 2020], a framework based on the specific Spring Cloud was designed and implemented to simultaneously deploy multiple versions of microservices in the computing environment. It extracts version information from source code to build version dependencies and routes user requests to the desired deployed microservice version at runtime.

In our work, besides modelling microservice-based applications with their version information, we also target the traceability of microservice evolution through transitions between configuration types, i.e. sets of microservice artefacts present in an implementation repository, and through transitions between configurations, i.e. sets of running microservices. In addition, we take into account two main microservice communication modes, not only client-server but also publish-subscribe. Another issue in the field of software evolution for microservices concerns safe dynamic updating and is still an open research area. It is the focus of this thesis.

2.3 Dynamic Software Updating

In the previous sections, we have presented an overview of concepts related to software architecture and software evolution. In this section, we discuss the case of dynamic software updating (DSU), which is considered as a specific subject in software evolution, including the following topics: the definition and requirements of DSU (Section 2.3.1), the characterisation of changes with their impacts (Section 2.3.2), the type of distributed application considered, i.e. the distributed application model (Section 2.3.3), an overview of existing approaches on when the updates can be performed safely (Section 2.3.4), the presentation of strategies to reach update conditions (Section 2.3.5), and an introduction to basic distributed algorithms for termination detection of service calls used in DSU (Section 2.3.6).

2.3.1 Objectives of DSU

DSU enables microservice updating at runtime in order to adapt to changing environments without, whenever possible, the shutdown and the restarting of the whole distributed application. DSU has been identified as a fundamental problem of distributed systems in the seminal work of Kramer and Magee [Kramer and Magee, 1990]. In [Miedes and Munoz-Escoı, 2012, Seifzadeh et al., 2013, Ahmed et al., 2020], the authors summarise a list of requirements and goals of DSU. We list below the ones that we have selected for our work in an industrial setting:

— *Continuity and Minimal Disruption*: Updating should not interrupt the execution of the software for a too long time. The best case is that the update process does not block any client service call execution nor impact performance.

— *Transparency*: Besides expected results, DSU should have no other significant impacts on its context, including final users, development teams or managed system. Concerning the end users, i.e. "user transparency", they do not need to know any details of the updating mechanism, nor the specific skills to perform the updating. Concerning the development teams, i.e. "programmer transparency", updating should not change the way of designing and developing the systems. Concerning the software itself, i.e. "application transparency", the update mechanism does not impose any constraints on how the programs are implemented and does not change any intended behaviour.

— *Generality*. A general update mechanism allows applying different types of updates and changes at different granularity levels (e.g. interfaces, connections) on the one hand, and updating heterogeneous system elements (e.g. different technologies, programming languages, models) on the other hand.

— *Consistency and Integrity*. Consistency means that the distributed application should be in the same state after the dynamic update as if the update were performed offline, i.e. the system is shut down, updated, and restarted. Integrity of dynamic updates means that the client service calls that are interrupted by DSU should be properly terminated, as the microservice states are valid.

— *Version Coexistence*. It should be possible that the old version and the new version of the same elements exist at the same time.

Some of the other requirements for DSU are *state preservation* (in our work, we ignore state transfer between microservice "versions"), *simultaneous updates*, ability to *schedule* (we consider semi-autonomic adaptation with engineers that explicitly request

when updating), ability to *roll back* in case of failure (our contributions assume that the system is fault-free and that does not consider DSU failures).

2.3.2 Characterisation of changes

Different kinds of changes can lead to different approaches for performing the dynamic updating. For example, breaking changes may always have wider impacts on the rest of the system, requiring more attention when updating. We present two ways to characterise the changes, one based on labels applied by developers during deliveries, namely SemVer (Semantic Versioning), and another one based on model changes (the structure and the behaviour) of the microservices, namely essential and non-essential changes.

Semantic versioning. Semantic Versioning (SemVer) [Preston-Werner, 2013] is commonly used in software development and versioning to control the configuration and the growth of version numbers [He et al., 2020, Ma et al., 2019]. This method introduces a set of rules and requirements on how to assign version numbers and whether a new version is backward compatible [Decan and Mens, 2019]. Considering the version format of `X.Y.Z` (`Major.Minor.Patch`), SemVer informs software architects and system administrators, and helps IT teams to anticipate potential breaking changes:

— Major (`X`): backward incompatible API changes that indicate that this version is not necessarily compatible with previous major versions,

— Minor (`Y`): backward compatible API additions or changes that indicate that this version is still compatible with previous minor versions,

— Patch (`Z`): bug fixes not affecting the API, where the backwards compatibility is assured with previous minor versions.

The goal of SemVer is to give each new software version a unique number with compatibility information. However, this version numbering presents limited structural compatibility semantics. When it comes to characterising changes in terms of behaviour changes, the SemVer approach is not sufficient.

Essential and non-essential changes. Another desired property of changes should take into account the structural and the behavioural part of the model of the microservices. This property depends on whether the possible execution of any future message in a collaboration is to produce the same resulting state and side effects before and after the update.

The authors of [Kawrykow and Robillard, 2011] defined non-essential differences to be low-level code changes as: (i) *"naturally cosmetic"*, (ii) *"generally behavior-preserving"*, and (iii) *"unlikely to yield further insights into the roles or relationships of the entities they modify"*. In their work, they consider code changes and documentation-related updates, such as trivial type updates, local variable extractions, rename-induced modifications, etc.

In the very recent work of [Sokolowski et al., 2022], the authors supplement the previous definition of essential and non-essential changes by application-specific analyses of whether the changes introduce semantic changes: the identification of essential and non-essential changes can also be improved and refined by the insights from the application and the developers. The work considers workflows executing on FaaS and *"an update of a component $c \in C$ from version v to v' is an essential change for a workflow instance I [...], if the possible execution of any future task t [of the workflow] on v' is not guaranteed to produce the same resulting state and side effects as executing t on v."* The authors demonstrate that leveraging the distinction between essential and non-essential changes can introduce more possibilities to perform safe and efficient updates when changes are not essential.

2.3.3 Distributed application model for DSU

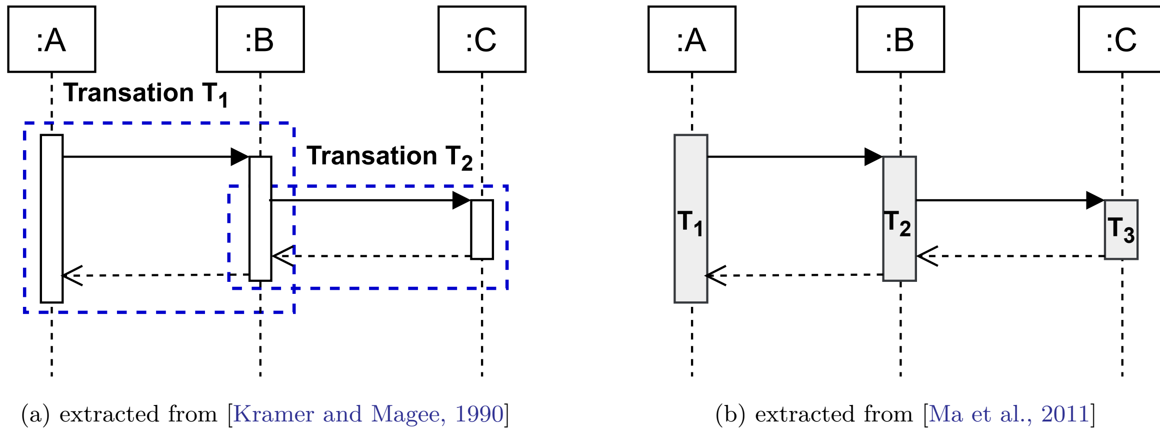
Most of the previous works on DSU consider synchronous distributed transactions in so-called component-based systems ([Kramer and Magee, 1990, Vandewoude et al., 2007, Ma et al., 2011, Baresi et al., 2017]). The work of [Sokolowski et al., 2022] considers asynchronous workflows, which are also translated to synchronous transactions by the authors for comparison. In order to provide a sound basis for a discussion on existing DSU solutions, we first present the involved concepts in this section, namely transactions and system consistency.

Definition of transaction. The notion of transaction is inevitably brought up in dynamic updates for distributed systems and contributes to understanding the consistent status of the system. It refers to a set of synchronous messages that are related and should be executed somewhat atomically. In [Kramer and Magee, 1990], the authors defined a transaction as follows: "*A transaction is an exchange of information between two and only two [participants], initiated by one of the [participants]. It is assumed that transactions complete in bounded time and that the initiator of a transaction is aware of its completion.*"

We can learn that a transaction consists of a sequence of messages and is initiated when a participant invokes a service from other participants, where the participant can

be a component, a client or other architectural elements.

Figure 2.3 shows two representations of transactions in a sequence diagram. In Figure 2.3a, a transaction is a sequence of horizontal arrows and a transaction relates two components, e.g. transaction T_1 relates to two components, namely A as an initiator and B as a receiver, and T_2 relates to B as an initiator and C as a receiver. In Figure 2.3b, a transaction is a UML activation bar: e.g. transactions T_1, T_2, and T_3 activated in components A, B, and C.



(a) extracted from [Kramer and Magee, 1990]

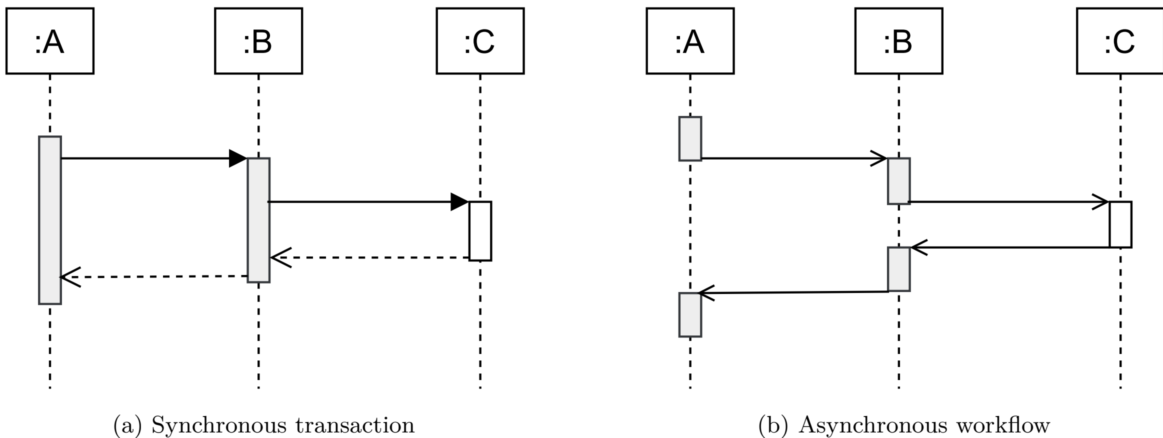
(b) extracted from [Ma et al., 2011]

Figure 2.3 – Examples of transactions in sequence diagram

In [Ma et al., 2011, Baresi et al., 2017], the transaction T_1 in both figures is called a root transaction and T_2 is a sub-transaction. In [Sokolowski et al., 2022], *workflows* implement complex and long-running transactions. Asynchronous workflows can emulate the synchronous transactions by splitting the parent transaction into two child transactions before and after the sub-transaction. The authors adopt the representation of transaction in [Ma et al., 2011, Baresi et al., 2017]. An example is shown in Figure 2.4, where the transactions in Figure 2.4a can be modelled as in Figure 2.4b. But then the granularity of the atomicity is not the same.

Application consistency. Application consistency is one of the most important properties to be preserved during DSU. The work of [De Palma et al., 2001] defined the application consistency in the case of a reconfiguration as follows: "*A distributed application is defined by a set of global distributed calculations. An application is consistent if the results of running calculations are not modified when a reconfiguration occurs.*" Consistency is evaluated on the states of the system and the correctness of running activities before and after updating a component in the system.

Moreover, in [De Palma et al., 2001, Ketfi et al., 2002], the authors make a distinction between local consistency and global consistency:



(a) Synchronous transaction

(b) Asynchronous workflow

Figure 2.4 – Examples of modelling transactions as workflow (adapted from [Sokolowski et al., 2022])

— *Local consistency* is concerned with the internal computation of a component, comprising neighbouring components, the local state, and the communication channels.

— *Global consistency* is concerned with the global distributed calculations in the whole system, referring to the collaborations of all components in the system.

The existing approaches to addressing system consistency fall into two major categories [Moazami-Goudarzi, 1999]:

— *Consistency through recovery*: It allows inconsistency to occur, but relies on recovery mechanisms to restore consistency after the update is complete. One example of the use of recovery for dynamic evolution is presented in [Feiler and Li, 1998], which focused on offline analysis to determine configuration inconsistencies and identify reconfiguration paths to recover to consistent configurations. This approach requires that the developers provide a rollback recovery mechanism in the application, which brings the work of recovery implementation to developers.

— *Consistency through avoidance*: It intends to prevent inconsistencies from occurring in the first place. The system waits for an ideal time called a *safe* state to execute the updates so that inconsistencies are not even introduced. Roughly speaking, this approach: (i) identifies the part of the system being affected and detect a safe state; (ii) deactivates the affected part of the system; (iii) performs the reconfiguration; (iv) reactivates the affected part and returns to normal operation. It helps to minimise system overhead, disruption, and programmer participation.

Our approach and the solutions that we will discuss in this thesis apply the second approach for ensuring consistency through avoidance. Therefore, we present in the next section the state of the art on the update conditions.

2.3.4 Consistent update conditions

The question of when an update can be performed consistently has been formulated in four update conditions that are referenced many times in component-based systems, three of which are safe, namely quiescence [Kramer and Magee, 1990], freeness [Ma et al., 2011, Baresi et al., 2017] and essential freeness [Sokolowski et al., 2022], the unsafe one being tranquillity [Vandewoude et al., 2007]. All these works define the update condition for the version change of only one component in a distributed system. We now present these four update conditions from the literature. All the definitions of the update conditions below are expressed using a model of the distributed application at runtime.

Quiescence. In [Kramer and Magee, 1990], a component is either active or passive. Here follow the definitions of these two statuses (we use the term *status* instead of the term *state*, where *state* refers to the internal state of a component and *status* refers to the conditions related to the DSU algorithm):

— "*A [component] in the* ***active*** *[status] can initiate, accept, and service transactions*".

— "*A [component] in the* ***passive*** *[status] must continue to accept and service transactions, but* (i) *it is not currently engaged in a transaction that it initiated, and* (ii) *it will not initiate new transactions.*"

The transitions between active status and passive status by the `activate` and `passivate` actions are described in the state machine diagram of Figure 2.5.

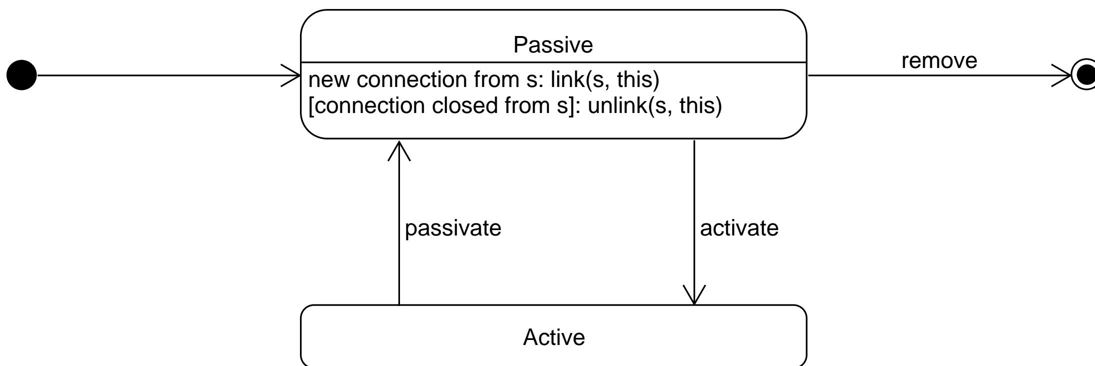


Figure 2.5 – State machine diagram of components (adapted from [Kramer and Magee, 1990])

The passive status contributes to system consistency by completing transactions. However, it is not sufficient for safe dynamic updating due to its possibility of processing

transactions initiated by other components. Then, the stronger *quiescence* property has been proposed and defined as follows: *"A [component] is **quiescent** if:*

1) *it is not currently engaged in a transaction that it initiated,*
2) *it will not initiate new transactions,*
3) *it is not currently engaged in servicing a transaction, and*
4) *no transactions have been or will be initiated by other [components] which require service from this [component]."*

According to the definition, a component in the quiescent status is considered both *consistent* and *frozen*. That is to say, the application state does not contain the results of partially completed transactions, and will not be changed by new transactions. A state machine diagram of the transition to quiescence is presented in Figure 2.6.

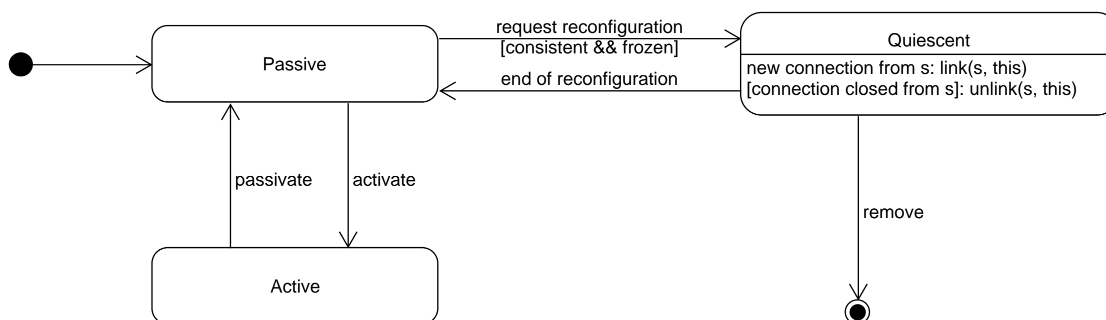


Figure 2.6 – State machine diagram with quiescence (extracted from [Kramer and Magee, 1990])

This quiescence property depends on the component itself as well as its directly or indirectly connected components. In order to reach the quiescent status of a component, the following conditions should be satisfied:

1) the component itself must be passive,
2) all components that can directly initiate a transaction on this component must be passive,
3) all components that can initiate root transactions that result in sub-transactions on this component must be passive.

The authors have proven that the quiescence update condition is reachable in bounded time, as long as each transaction terminates within a bounded time and deadlocks are avoided. Quiescence is a sufficient condition for safe and consistent dynamic updating. However, it may lead to a significant disruption of the running application for even a small update, because placing a component in a quiescent

status may require to passivate many other components (all the directly or indirectly connected components).

Tranquillity. In order to reduce the disruption of quiescence, in [Vandewoude et al., 2007], the *tranquillity* property is defined as follows: "*A [component] is in a* ***tranquil*** *status if:*

1) *it is not currently engaged in a transaction that it initiated,*
2) *it will not initiate new transactions,*
3) *it is not actively processing a request, and*
4) *none of its adjacent components are engaged in a transaction in which it has both already participated and might still participate in the future.*"

Note that regarding the black-box principle, all participants in a transaction either are the initiator or are directly connected (also called *adjacent*) components. In other words, the existence of indirectly connected components and of sub-transactions is unknown to the initiator.

In some cases, it is possible that the component to be updated will never reach the tranquil status. Such a situation is given in [Vandewoude et al., 2007]: the component to be updated is used in an infinite sequence of interleaving transactions. If the tranquillity is not reached, then an implemented fallback mechanism is applied to return to quiescence. The transition between the status of a component is presented in the state machine diagram of Figure 2.7.

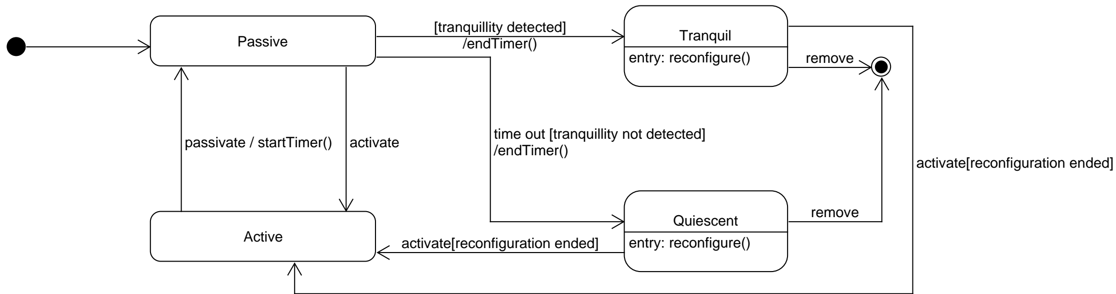


Figure 2.7 – State machine diagram of the transition to the tranquillity property

The tranquillity condition has better update timeliness and less interruption than quiescence. But it is not a safe property if the black-box principle is not respected. Figure 2.8 shows an example of a possible unsafe updating that is permitted by the tranquillity property. Transaction T_1 is a root transaction. T_2 and T_3 are sub-transactions of T_1. T_4 and T_5 are sub-transactions of T_3, so that they are "sub-sub-

transactions" of T_1. According to the black-box principle, T_4 and T_5 are not aware of the existence of T_1. Then, let look at component B at time t_3. At t_3, B has already completed T_2 and has not yet started to participate in T_4. In the view of A, B will no longer participate in other transactions in the future. So, B is said to be tranquil at time t_3. However, updating B at t_3 is not safe because T_2 has been executed with the old version of B and T_3 will be executed with the new version at t_4. If the two versions of B are semantically related (e.g. encryption and then decryption), the execution of T_4 might not get the expected result.

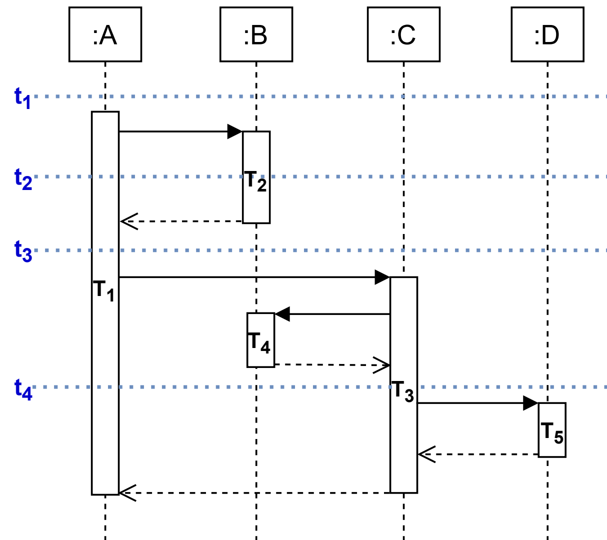


Figure 2.8 – Example illustrating several activation periods of the same component (T_i: transaction, t_i: time)

In addition, tranquillity only ensures local consistency, but not global consistency. To address this shortcoming, in [Ghafari et al., 2012], the authors proposed an additional "serenity" status. But since these update conditions are not safe when the black-box principle is not used, we do not consider them further in this thesis.

Freeness and version consistency. In [Ma et al., 2011, Baresi et al., 2017], the authors proposed "*version consistency*" as a sufficient global condition for the safety of dynamic updating, and the corresponding local update condition is called "*freeness*". A dynamic update is **version consistent** if a transaction is entirely executed in the same configuration: either the one before the change or the one after the change. A component is said to be **free** if:

1) it is not hosting/executing any transactions, including the root transactions that it initiated and all its direct and indirect sub-transactions,
2) all the transactions involving this component have already been executed,

3) there does not exist any transaction to be initiated on this component in the future.

First of all, note that, compared to quiescence, freeness requires the knowledge of the graph of transaction calls: the set of transactions in the future is calculated by knowing which ongoing transactions may lead to a sub-transaction on the component. This is why the authors of [Ma et al., 2011, Baresi et al., 2017] model the static dependencies (links) and the dynamic dependencies (interactions) of the distributed application. The dynamic dependencies are temporal relationships (past or future) between components for ongoing computations and are manipulated at runtime. Therefore, freeness is checked with the help of the dynamic dependencies on the component that is to be updated, along with the direct and indirect sub-transactions of the component.

In the example of Figure 2.8, component B is free at times t_1 and t_4: at t_1, B is quiescent, and at t_4, B has participated and will no more participate to root transaction T_1. Component B is not free at time t_2 because it is active and, at time t_3, because potential transaction T_4 on B may execute in the future.

Essential freeness and essential safety. The work presented in [Sokolowski et al., 2022] considers asynchronous workflows. The authors introduced *essential freeness* by distinguishing between different types of updates: essential and non-essential changes. In case of a non-essential change, the update of a component is safe as soon as the component is passive. In case of an essential change, it returns to freeness. "*A component is* ***essentially free*** *if:*

1) *it is not active,*

2) *[either] a. it will not be active in a workflow instance in which it already executed a task,*
   [or] b. its update is a non-essential change for all workflow instances in which it already processed a task."

Finally, an update guarantees *essential safety* only when the components that change versions are essentially free. Essential safety is equivalent to version consistency if all the updates are essential changes, and version consistency is a conservative *over-approximation* of essential safety. In the example of Figure 2.8, component B can also be updated at time t_3 if the change is non-essential.

2.3.5 Reaching strategies

In order to reach and uphold the update condition, the following four reaching strategies are being staged [Ma et al., 2011, Baresi et al., 2017, Sokolowski et al., 2022]:

— *Waiting*: The system just waits for the update conditions to manifest themselves. It is possible that the update condition of a component might never be reached, for example when there are always transactions running on it.

— *Blocking messages*: The starting of sending messages to the component to be updated is delayed until after the update. Deadlocks may appear because of the concurrent execution of two transactions that are hosted on the same component and need to access some shared resources.

— *Blocking transactions or workflows*: Instead of delaying message sending, the starting of new root transactions or of new workflows that may involve the component is delayed until after the update.

— *Concurrent versions*: This strategy lets two versions of a component co-exist during the update. The old version remains available until no transaction or workflow needs it anymore.

Since quiescence relies only on static information, the authors of [Kramer and Magee, 1990] can only discuss the "blocking transactions" strategy by passivating a set of components, ensuring that no transaction will be invoked on the components in the future. The work of [Vandewoude et al., 2007] is generally unsafe when using the "blocking messages" strategy because of the possibility of deadlocks in transactions. Thus, what they use is the "waiting" strategy, and if the tranquillity is not achieved, they resort to "blocking transactions". The work of [Ma et al., 2011, Baresi et al., 2017] discusses all the strategies and the authors prefer "concurrent versions" whenever applicable, and then "blocking messages". The work of [Sokolowski et al., 2022] discusses all these four strategies. Recall that when the update is an essential change, essential safety is the same as version consistency.

2.3.6 Distributed termination algorithm

Many terms of the different update conditions are expressed as the detection of the end of a client service call. Thus, the basis of any solution is a distributed termination algorithm [Tel, 2002]. By analogy with the distributed termination problem, a service call is terminated when all the architectural elements that may be involved are passive (not running) and no application messages about that service call are in transit to restart a passive microservice.

The literature distinguishes two families of distributed termination algorithms:

— Dissemination-based, also called computation-based, algorithms [Francez, 1980, Dijkstra and Scholten, 1980]: These algorithms complement the application algorithm with control messages. For instance, the algorithm of [Dijkstra and

Scholten, 1980] builds a tree of the application algorithm and detects termination when the tree is empty. Therefore, the message complexity of the termination of a transaction is the message complexity of the transaction. As a consequence, for long-running transactions that involve many messages, the termination is expensive.

— Wave-based, also called snapshot-based, algorithms [Chandy and Lamport, 1985, Dijkstra, 1987]: These algorithms repeatedly broadcast a wave to check whether a condition is satisfied in each involved component. Therefore, the message complexity of the termination of a transaction is the message complexity of the wave, regardless the complexity of the running transactions.

To the best of our knowledge, the solutions of the literature (of [Ma et al., 2011, Baresi et al., 2017, Sokolowski et al., 2022]) are all dissemination-based: Every transaction or workflow is complemented with control messages, whether in periods without updating or during updating. These algorithms are optimal in the worst case and they require the channels to be bidirectional and FIFO. In contrast, our contribution proposes a solution based on snapshots. Indeed, we intend to limit the overhead of our solution in terms of the exchange of messages. We also leverage the behaviour of a DSU algorithm, which is based on phases. These phases naturally delimit global system states, i.e. distributed snapshots.

2.4 Conclusion

In this chapter, we have briefly introduced the background and concepts for understanding our work in this thesis.

First, we have outlined the history of microservices, i.e. where did microservices come from and why this architectural style emerged, and also presented their definitions and characteristics. We argued that one of the most interesting properties is the modularity for their development, deployment and evolution.

In this thesis, we are interested in the research area of software evolution of microservice-based applications. Thus, we have provided an overview of software evolution, and focused on runtime evolution, where systems can adapt to changes dynamically and evolve during execution. We have highlighted that software architectures, e.g. microservice architectures, play a critical role in runtime evolution. In our work, we apply the principle of the MAPE-K control loop for automating our evolution tasks. In order to control and trace changes, we follow the model at runtime approach for the representation of microservice-based applications and their version management, which acts as the Knowledge base in the MAPE-K loop. Considering

the degree of automation, our proposal can be regarded as semi-automatic: architects and developers analyse the documentation or UML model plus the code to extract the required information for building the knowledge base, and the planning and execution are automatically performed.

Dynamic software updating (DSU) is one of the specific topics in software evolution. DSU enables microservice updating at runtime and should respond to several requirements, such as continuity of service, independence from the context impacts or system consistency. Some assumptions are made for DSU in our work: (i) we ignore state transfer between microservice versions, and (ii) our managed system is fault-free. We have then presented two ways to characterise the changes, which will both be used in our solution. In our work, we follow the policy of SemVer to express the version numbers for elements that need to be versioned in microservice architectures, and we complement SemVer with the classification into essential or non-essential changes by considering the behavioural aspects of the microservice model.

In addition, we have presented the state-of-the-art DSU solutions that respond to the question of when an update can be performed safely and consistently. All these solutions target global consistency (of the whole distributed application) and address system consistency through avoidance, i.e. waiting for a safe status (update condition) before updating. Our solution also follows this approach. In order to reach and uphold the update condition, the reaching strategies and algorithms have also been presented in this chapter. Finally, note that the basis of the solutions is a distributed termination detection algorithm. All the previous DSU algorithms of the literature are dissemination-based, while our contribution proposes a novel snapshot-based approach.

In the next two chapters, we will detail our two main contributions about the modelling and the dynamic evolution for microservice architectures.

Chapter 3

Runtime Models and Evolution Graph for Version Management of Microservices

Contents

3.1 Case Study GDE **37**
3.2 Runtime Models **38**
3.2.1 Model of types 39
3.2.1.1 Configuration types view 39
3.2.1.2 Microservice types view 40
3.2.1.3 Contract types view 42
3.2.2 Model of instances 46
3.3 Evolution Graph **48**
3.3.1 Overview of the graph building process 49
3.3.2 Illustrative scenarios 50
3.3.2.1 Patch change 52
3.3.2.2 Minor change 53
3.3.2.3 Major change 54
3.4 Implementation in MIMOSAE **55**
3.4.1 PDDL planner 56
3.4.2 Executor 59
3.5 Discussion **60**
3.6 Conclusion **62**

In this chapter, we present our first contribution: building runtime models and evolution graphs to (i) help engineers manage microservice version management, and (ii) abstract architectural evolution in order to manage reconfiguration deployments. More precisely, these requirements can be refined into the following typical requirements identified in industrial contexts: (1) in order to rationalise the cost of evolution and maintenance of heterogeneous software solutions, a global and unified view of the ecosystem evolution should be provided to architects, developers, and administrators, and (2) the evolutionary changes to the system should be traceable and rolled back in case of an invalid configuration or any other abnormality.

In Section 3.1, we introduce an illustrative application involving synchronous (RPC calls or transactions) and asynchronous (publish-subscribe messages or events) interactions within an information system. The distributed application is extracted from an industry project and used to demonstrate our proposal.

In Section 3.2, we propose runtime models to represent the essential elements of microservice architectures into two views: a microservice type model and a microservice instance model. The microservice type model describes a structural abstraction of microservice architectures and the instance model captures their specific deployment configurations, which themselves conform to the type model. In addition, these models support publish-subscribe communication in addition to client-server communication.

In Section 3.3, in order to allow the traceability of microservice versions and their deployment, we build up an evolution graph into two parts: a sub-graph of configuration type snapshots for tracing the software evolution of software artefacts (types) and a sub-graph of configuration instance snapshots for tracing deployments of microservices (instances), the two sub-graphs being linked through the relation “an instance conforms to a type”.

In Section 3.4, we give an overview of the implementation of the proposal. It is based on a model written in JAVA, reconfiguration plans constructed with an AI planner, and the executor written in Go and using the Docker and Kubernetes APIs for executing reconfiguration actions. The implementation is demonstrated with the case study presented in Section 3.1.

3.1 Case Study GDE

To illustrate our contributions, we use a microservice-based application from the EDF Labs context: GDE stands for *Gestionnaire de Données d’Études* in French. This is a Scientific Research Data Management System. The global structure is displayed in Figure 3.1. The main objective of this information system is to characterise scientific

research data that are produced or used by researchers and engineers during industrial simulation activities. This application consists of six microservices. The `Project Service` and `File Service` manage scientific simulation projects metadata and related research data. The `User Service` manages the information on application users and the `Permission Service` controls their access permissions. The `Authentication Service` verifies the users' login and password via Single Sign-On and the `Logger Service` records login history. Each Microservice has its own database and communicates with others through either synchronous HTTP REST or asynchronous publish-subscribe interactions.

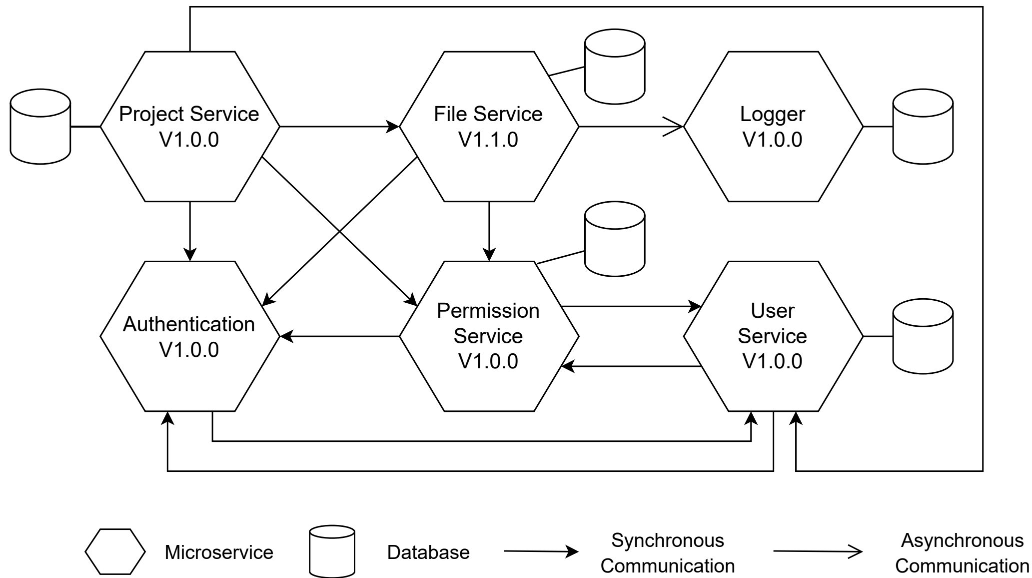


Figure 3.1 – A global view of GDE illustrative application

3.2 Runtime Models

In this section, we present the models that we build at design time and use at runtime for abstracting the microservice architecture and its versioning. We use model at runtime concepts [Blair et al., 2009] to provide a unified representation of up-to-date elements in the system driving evolutionary changes. Our proposed model is divided into two parts: the type model (Section 3.2.1) and the instance model (Section 3.2.2). The type model captures the structure of instantiable elements of the evolving microservice application, and the instance model captures the replicas of the corresponding deployable elements of the managed system.

3.2.1 Model of types

The model of types plays the role of providing a configuration specification for the system structure. All types that may be instantiated in the future deployment are added to this model, that is, (1) which types of microservices can be contained in a microservice-based application and in which version, (2) which interfaces can be provided or required by each microservice, and (3) how these microservices can be connected. In the following, we detail the different levels of data granularity of our model of types, from top to bottom, including three main views: configuration types view (Section 3.2.1.1), microservice types view (Section 3.2.1.2), and contract types view (Section 3.2.1.3).

3.2.1.1 Configuration types view

The root class of the model of types is named `ConfigurationType`. As its name suggests, a configuration type aggregates all the types that the configuration of a managed microservice-based application may contain. As shown in Figure 3.2, these are the types that are available, for example, as code artefacts or as container images in the implementation repositories: i.e. available microservice types (class `MicroserviceType`), available connector types (class `ConnectorType`), available contract types (class `ContractType`), and available database system types (class `DataBaseSystemType`).

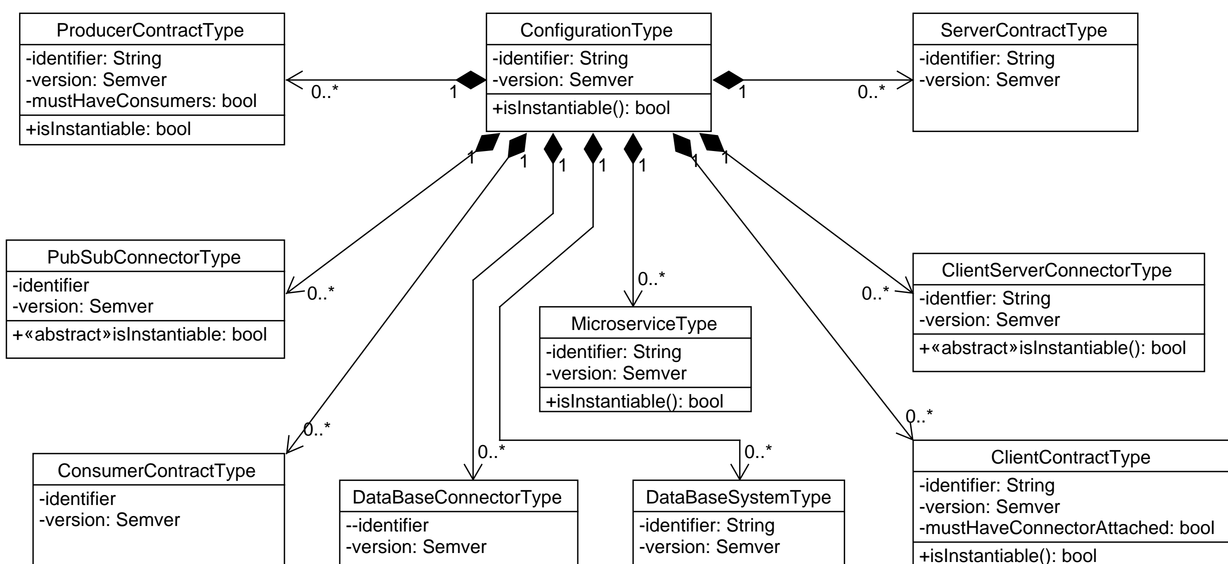


Figure 3.2 – Model of types: configuration types view

We follow the policy of Semantic Versioning (SemVer) [Preston-Werner, 2013] in our models to express the version numbers of all the type elements that need to be versioned in microservice architectures. The format of a version number is X.Y.Z (Major.Minor.Patch). Naturally, all the types are uniquely identified by a name and a version, e.g. a microservice type authentication_service in version 1.0.0 or a publish-subscribe connector type rabbitmq in version 3.9.1. This identification of types is represented in the class diagram of Figure 3.2 by the two attributes identifier and version of each type. Other class attributes and methods represented in the diagram will be introduced in the next section.

3.2.1.2 Microservice types view

The central concept of the model of types is the microservice type. Figure 3.3 provides the view that is centred on the microservice type (class MicroserviceType). Each microservice type exposes a set of contract types, which are the "interfaces" provided or required by an instance of the microservice type. These contracts are intermediate entities to connectors [Mehta et al., 2000] and are where architects specify the quality of service of the connections, e.g. for producer and consumer contracts [Lim et al., 2015]. Considering different possible ways to set up communication in a microservice-based application, we distinguish between synchronous interaction in client-server mode (classes ClientContractType, ServerContractType, and ClientServerConnectorType) and asynchronous interaction in producer-consumer mode (classes ProducerContractType, ConsumerContractType, and PubSubConnectorType). Even though synchronous connectors may be skipped in some way, the concept of connector for asynchronous communication becomes more significant. We therefore keep it for both interaction modes.

Synchronous channels, using for example HTTP-based REST communication, are represented by client-server connector types. A client-server connector type brings together a client contract type and a server contract type, where the multiplicity is set to 1 on the side of ServerContractType and ClientContractType. The mustHaveConnector-Attached attribute in the client contract type indicates whether the connection to a server microservice is mandatory, i.e. whether the client may operate in degraded mode without a connection to a compatible server. In Figure 3.1 of the GDE case study, microservice type authentication_service provides server contract types to others, and requires a client contract type that is connected with a server contract type from microservice type user_service through a client server connector type.

In the context of this thesis, asynchronous communication is limited to publish-subscribe systems. Some other asynchronous modes, such as message streaming, are not discussed in this thesis. Therefore, asynchronous channels are represented by the

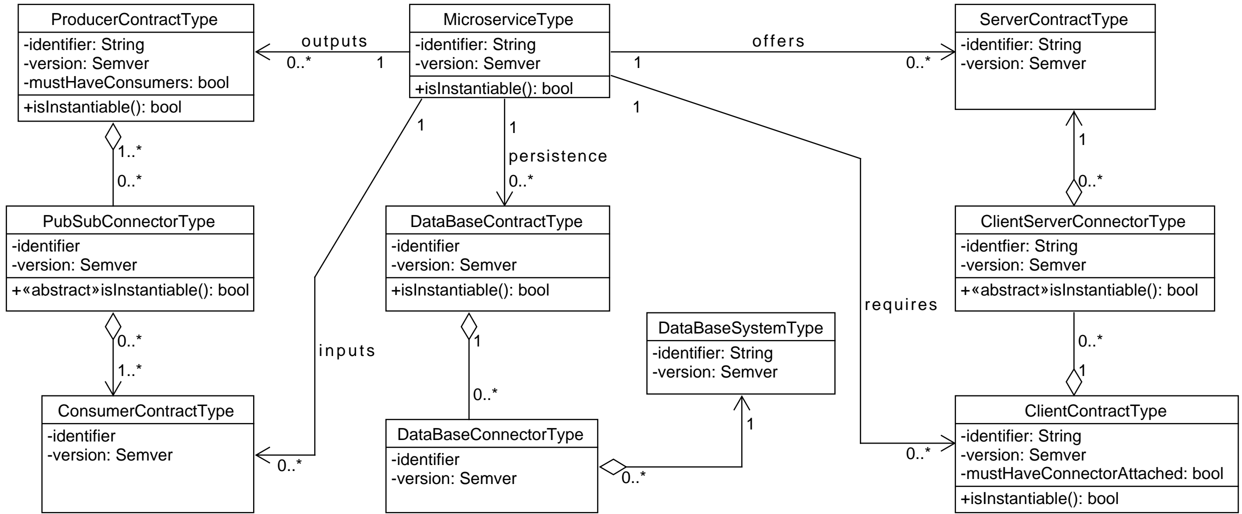


Figure 3.3 – Model of types: microservice types view

publish-subscribe connector types, typically using message broker technologies, e.g. MQTT [OASIS, 2019] and AMQP [AMQP Consortium, 2010] brokers. A publish-subscribe connector type brings together a collection of producer contract types and a collection of consumer contract types, so that, the multiplicity is set to `1..*` on the side of `ProducerContractType` and `ConsumerContractType`. Producers post messages to an intermediary message broker, and consumers register subscriptions with that broker, letting the broker perform the filtering. Providers are not aware of the consumers. Based on different common forms of filtering, we differentiate channel-based, topic-based and content-based publish-subscribe systems [Eugster et al., 2003, Mühl et al., 2006], the first two of which will be detailed in the next sections. A similar `mustHaveConsumers` attribute given in producer contract type indicates whether the logical link to some consumer microservices via a broker is mandatory, i.e. whether the producer microservice can operate in degraded mode with no consumers with a subscription filter matching the routing key of its messages. In the GDE case study, the `file_service` microservice type publishes its logs, which are forwarded to the `logger_service` microservice type (because the logger accepts all the logs).

Another category of connector types is the database connector types. Database contract types are where architects specify the connections to database systems. Every declared database contract types of a microservice type is mandatory. Taking the example of the GDE application in Figure 3.1, microservice type `authentication_service` does not have any database system type. On the contrary, microservice types `user_service`, `permission_service`, `project_service` and `file_service` have their own SQL database

system type `PostgreSQL` in version 14.5.0, and microservice `logger_service` has a NoSQL database system type `Redis` in version 7.0.4.

As mentioned earlier, all of these types of microservices, database systems, contracts and connectors constitute a configuration type of the application so that any version change of a microservice creates a new microservice type with the same identifier but a changed version. It is up to the software architects to decide which microservice types with which versions can be included in the current configuration type.

Every time a change is committed in the type model, the instantiable property is checked by the `isInstantiable()` method to verify whether the current configuration type is valid and ready to be instantiated. We define that a configuration type is instantiable, namely an instance model can be created that conforms to these types if and only if all the microservice types in this configuration type are instantiable. The property "instantiable" is defined using the property "compatible". The property "compatible" is formalised and defined later in Section 3.2.1.3. For the sake of comprehension, intuitively, a client contract type is compatible with a server contract type if all the operation declarations of the client contract type can be found in the server contract type; a producer contract type is compatible with a set of consumer contract types if all the event types produced are accepted by one of the corresponding consumer contract types and their filtering matches (this will be detailed later according to different filtering mechanisms). Thus, concretely, microservice types are instantiable if: (i) client contract types are compatible to attached server contract types, (ii) client contract types that must have connectors attached are indeed linked, (iii) producer contract types are compatible with connected consumer contract types, (iv) producer contract types that must have consumers actually have consumers, (v) all the database contract types are indeed connected to database systems via database connectors.

A part of the example of applying our GDE case study to the model of types is presented in Figure 3.4. This object diagram includes microservice types of `authentication_service` and `user_service`, along with their contract types and connector types that link these two microservice types.

3.2.1.3 Contract types view

Contracts of microservices always exist in order to achieve effective communication, operated by either contract owners or contract consumers. According to the state of practise in microservice-based applications for different mechanisms of synchronous and asynchronous communication pattern, we select client-server and publish-subscribe patterns, and distinguish them in the contract type view. The two are respectively introduced in the following.

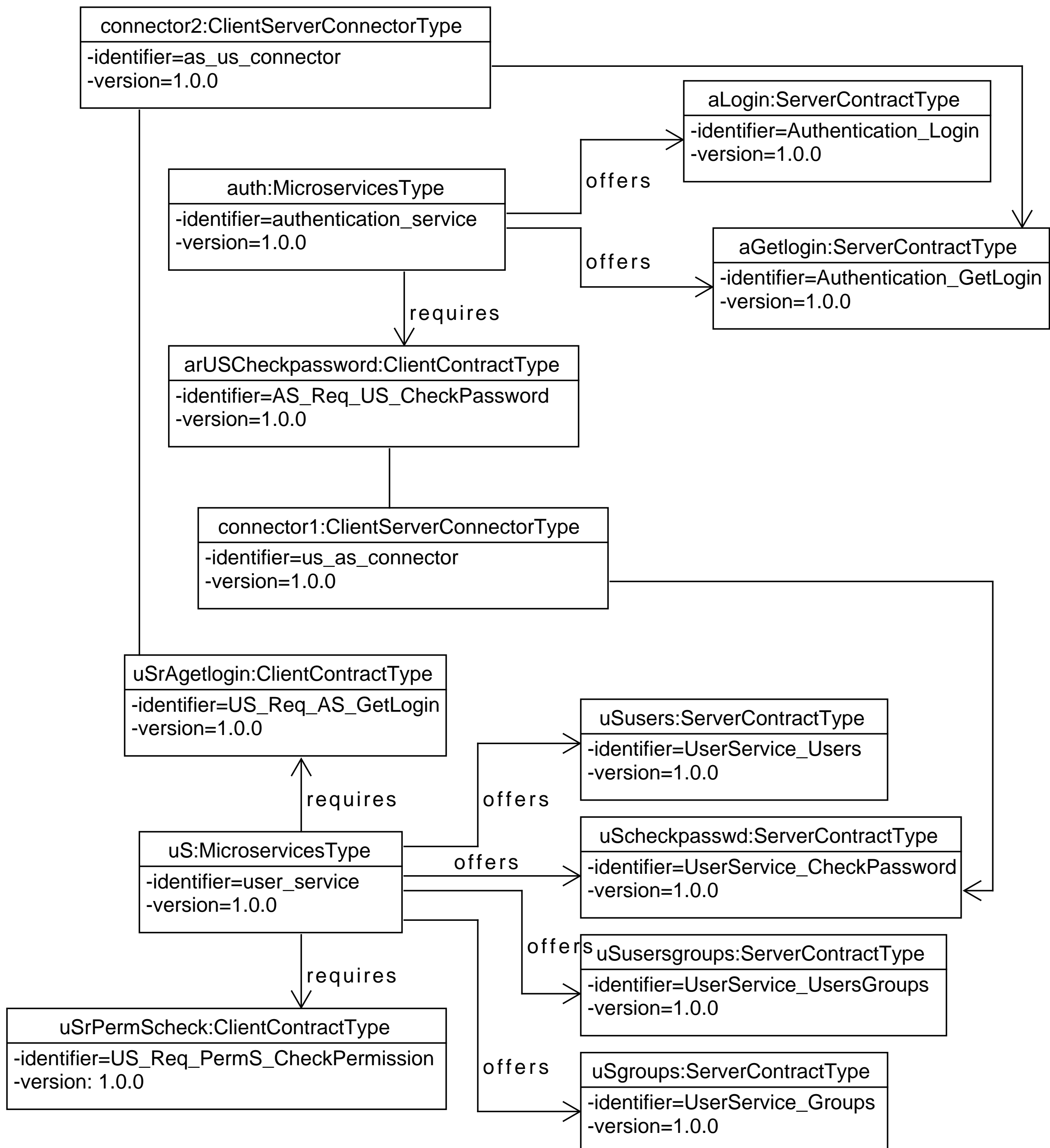


Figure 3.4 – Object diagram for the model of types applied to GDE: a part of microservice types view, including `authentication_service` and `user_service`

Client-Server contract types view. As depicted in Figure 3.5, server contract types and client contract types contain a collection of operation declarations, which include an operation name, a list of arguments, and a return type, i.e. $op = (name, args, rt)$. The type of arguments and the return type are either primitive types or object types. By definition, considering invariance in static type checking [Beugnard et al., 1999], two operation declarations are compatible with each other if and only if they have the same name, the same argument list, and the same return type, formally, $isCompatible(op_1, op_2) \stackrel{\text{def}}{=} (op_1.name = op_2.name) \wedge (op_1.rt = op_2.rt) \wedge (\forall k, 1 \leq k \leq |args|, op_1.args[k] = op_2.args[k])$. Then, we can define that a client contract type is compatible with a server contract type if and only if all the operation declarations of the client contract type are present in the server contract type. Formally, $isCompatible(cct, sct) \stackrel{\text{def}}{=} \forall op_c \in cct.ops, \exists op_s \in sct.ops, isCompatible(op_c, op_s)$, where cct and sct means respectively a client contract type and a server contract type that contain a set of operation declarations ops.

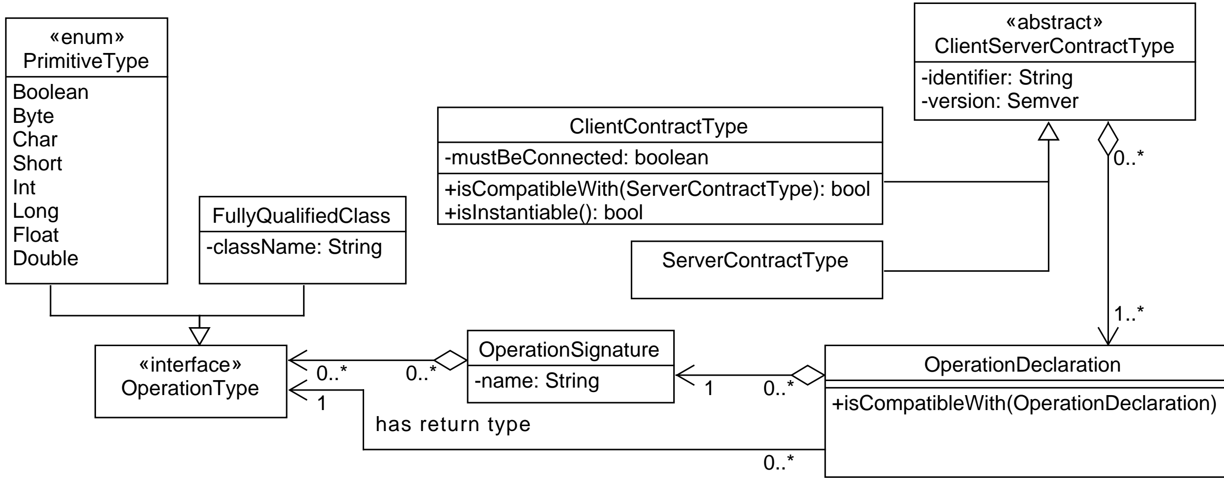


Figure 3.5 – Model of types: client-server contract types view

Publish-Subscribe contract types view. The process of selecting messages for consumption and processing is called filtering in the publish-subscribe pattern. Based on message filtering mechanisms [Mühl et al., 2006], the publish-subscribe model divides the producer contract types and the corresponding consumer contract types into three categories: channel-based, topic-based, and content-based, as shown in Figure 3.6.

Channel-based filtering is the simplest form of identifying sets of event types. As presented in Figure 3.7a, producers select a named channel into which an event is published, consumers also select a channel, and they will get all events published in

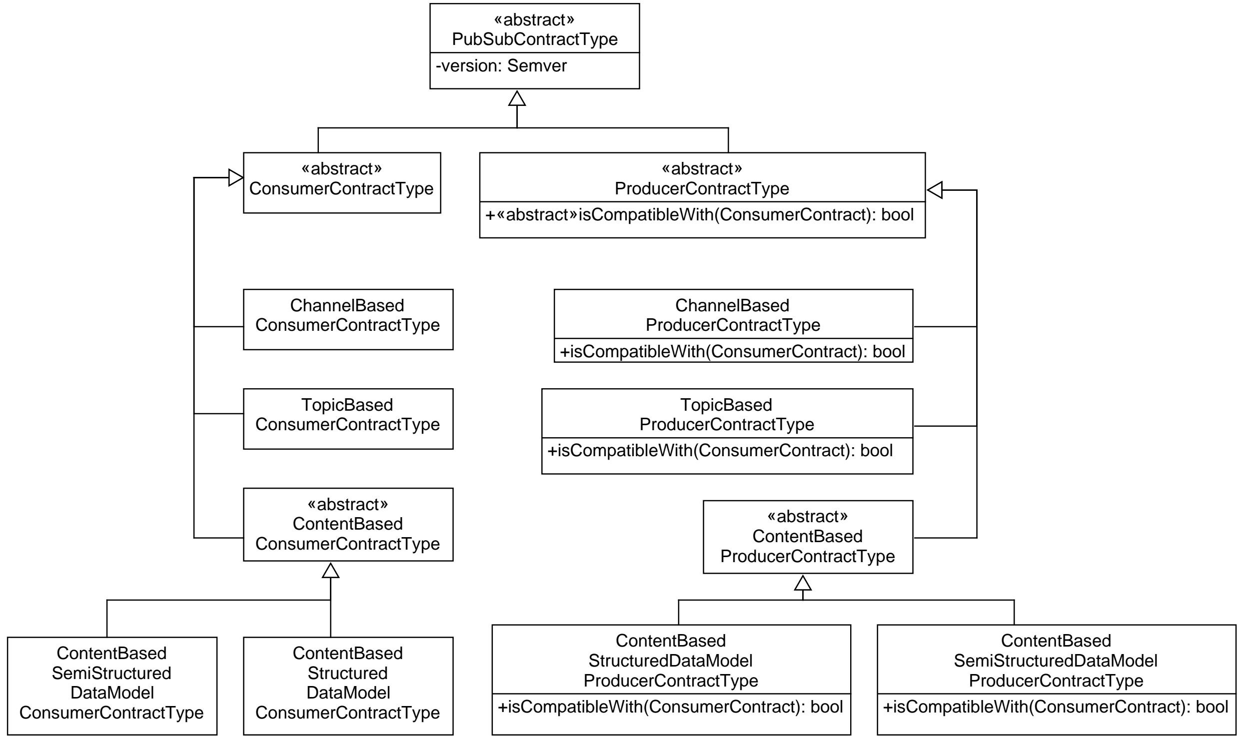


Figure 3.6 – Model of types: publish-subscribe contract types view

it. We can thereby define the compatibility of a producer contract type with a set of consumer contract types. It is important to note that the compatibility of a producer contract type is not asserted with a consumer contract type, but with a set of contract types, i.e. the whole set of produced event types must be consumed, but it is not necessary that the consumption is carried out as a whole by a single consumer contract type. Considering channel-based filtering, a producer contract type is compatible with a set of consumer contract types if and only if the channel names are equal and all the event types produced are accepted by one of the consumer contract types. Formally, $isCompatible(pct, cocts) \stackrel{\text{def}}{=} \forall evt_p \in pct.evts, \exists coct \in cocts, \Big(pct.channel = coct.channel\Big) \wedge \Big(\exists evt_c \in coct.evts, evt_p = evt_c\Big)$, where *pct* and *coct* means respectively a producer contract type and a consumer contract type that contain a channel (identified by its name) and a set of event types *evts*.

Topic-based filtering (a.k.a. subject-based filtering) uses string pattern matching for event selection, which is currently the most used by IT industry for application-layer distributed-event based systems. As shown in Figure 3.7b, providers produce topics that consumers can subscribe to, and consumers can receive all events that are published to matching topics they subscribe to. In other words, an event type sent with a particular

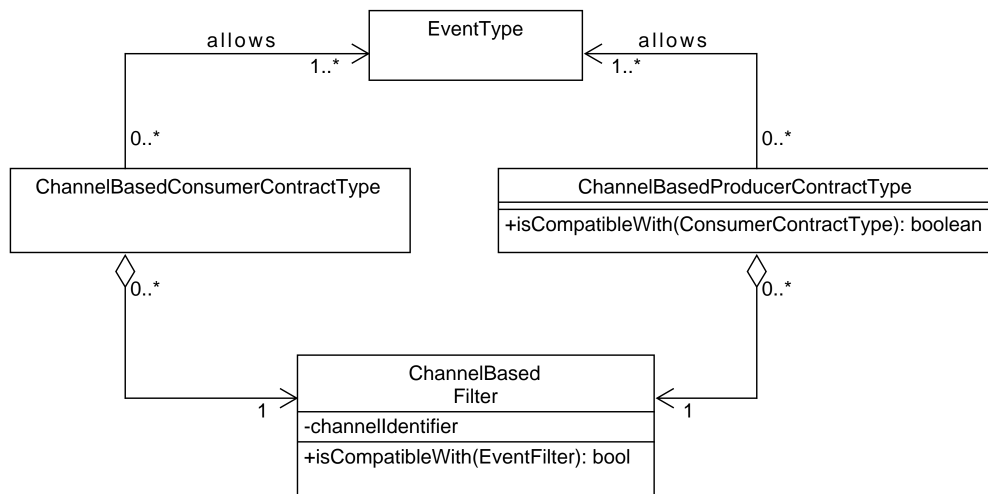
topic (in MQTT terminology) or routing key (in AMQP terminology) will be delivered to all objects that are bound with a matching topic filter (in MQTT terminology) or binding key (in AMQP terminology). Then, it follows that a producer contract type is compatible with a set of consumer contract types if and only if its topic or routing key (resp. in MQTT and AMQP) matches one of the topic filters or binding keys (resp. in MQTT and AMQP), and all the event types produced are accepted by the corresponding consumer contract types. Formally, $isCompatible(pct, cocts) \stackrel{\text{def}}{=} \forall evt_p \in pct.evts, \exists coct \in cocts, \Big(coct.filter(pct.topic) = true\Big) \wedge \Big(\exists evt_c \in coct.evts, evt_p = evt_c\Big)$, where *pct* means a producer contract type that contains a topic (a string decomposed into words) and a set of event types that can be produced, i.e. $pct = (topic, evts)$, and *coct* means a subscription filter (a string decomposed into words, which can be jokers) and a set of event that can be consumed, i.e. $coct = (filter, evts)$.

Another way of filtering is content-based, *a.k.a.* property-based, filtering. This filtering mechanism improves on topics by introducing a subscription scheme based on the actual content of the considered events [Eugster et al., 2003]. Some examples of solutions are template matching, extensible filter expressions on attribute/value pairs, XPath expressions on XML schemas [Cugola and Margara, 2012, Etzion and Niblett, 2011]. They can be categorized into content-based semi-structured and structured data models (see Figure 3.6). This last filtering mechanism is much more expensive and more adequate for complex event processing and streaming, which is not considered in this thesis.

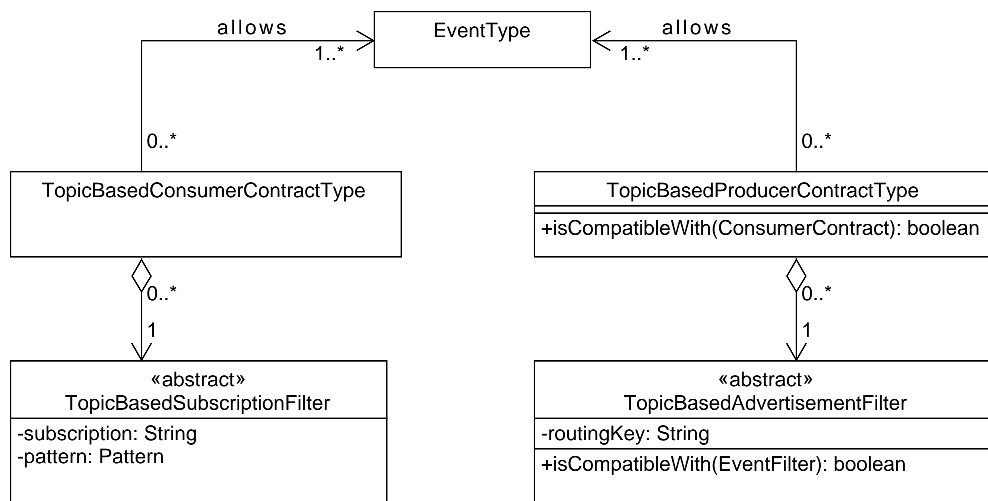
3.2.2 Model of instances

A part of the model of instances is shown in Figure 3.8, which can be considered as a specialization of microservices types view (presented in Figure 3.3). The grey elements in the figure come from the model of types. Each instance in this model of instances conforms to a type in the model of types. The root class is named `Configuration` and conforms to a `ConfigurationType`. A configuration is composed of a set of microservice instances, connector instances, and database system instances. A microservice type may have multiple microservice replicas that represent deployed microservice instances (class `Microservice`). Similarly, a database system type may have multiple instances, and a publish-subscribe connector type may also have multiple instances. Microservice instances are created from the microservice type, i.e. the type acts as a template such that, for each contract type of the microservice type, a contract instance is created.

In the model of types, the contract types and database system types are created first; then, microservice types and connector types are created. In the model of instances,



(a) Channel-based



(b) Topic-based

Figure 3.7 – Publish-subscribe contract types view: channel-based and topic-based

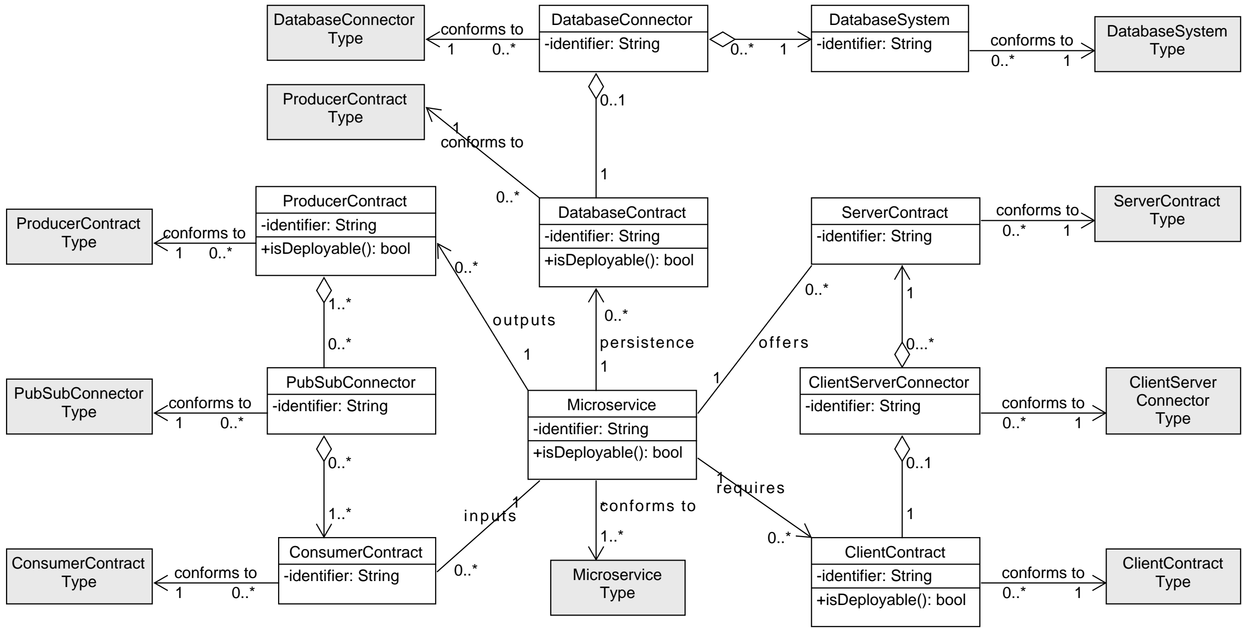


Figure 3.8 – Model of instances

database systems and connectors are created first; then microservices with their contracts are instantiated; and finally, microservice contracts are linked to connectors or database systems.

Whenever there are changes occurring in the current configuration, architects can check whether the instances are deployable. A microservice is deployable if and only if (i) its client contracts are deployable, i.e. the contract is attached to a connector when its type specifies that it must have a connector attached, (ii) its producer contracts are deployable, i.e. the producer contract has at least one consumer when its type specifies that it must have consumers, (iii) its database contracts are deployable. The property is checked by the `isDeployable()` method every time a change is committed in the instance model. Therefore, a model of instances represents the topology of the managed system with deployable microservices, connectors and database systems.

3.3 Evolution Graph

In order to trace the evolution trajectory of a microservice-based application, we propose an evolution graph, also from the two points of view of types and instances, that is, based on the runtime models of types and instances. Configuration types and configurations are organised in an evolution graph as depicted in Figure 3.9. We record the trajectory and history of how the system evolves over time at the granularity of a

configuration, which includes a set of microservices, not just for a single microservice. Each node in the evolution graph corresponds to a snapshot of the current system configuration as version changes are applied. These snapshots are photos “à la Git” that trace software evolution.

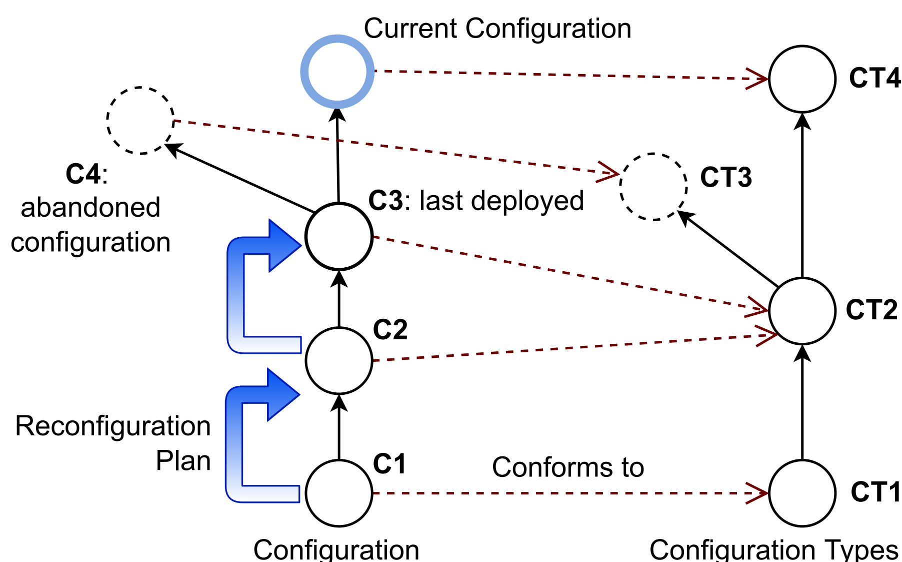


Figure 3.9 – Evolution graph of configuration snapshots and configuration types

3.3.1 Overview of the graph building process

The main process of applying a sequence of version changes by building the evolution graph is described as follows. When version changes are adopted in the code base, architects take a sequence of actions based on changes in the model of types, such as adding a new microservice type or a new connector type into the current configuration type. This creates the next configuration type to commit. A configuration type can be committed only if it is instantiable. This means that some errors can be detected and avoided at design time when committing a configuration. Once the commit of the configuration type is made, a new node representing a valid configuration type is created in the evolution graph (see circles with solid line in the right branch of configuration types in Figure 3.9 such as the node `CT4`).

Regarding the configurations from the instance model, each node corresponds to a configuration and has a unique link to a configuration type, i.e. to the configuration type it conforms to. Every configuration node, except the current configuration in preparation (see circles with a thick blue line in Figure 3.9 such as node `Current Configuration`), represents a deployable configuration, and each commit creates a

new node in the evolution graph (see circles with a solid line in the left branch of configurations in Figure 3.9 such as node C3).

The evolution graph of Figure 3.9 possesses several branches, one of which is for the abandoned nodes represented by dotted circles, e.g. configuration type CT3 is abandoned and configuration C4 conforming to CT3 is also abandoned. In some specific situations such as security issues, it makes possible to mark the abandoned nodes as obsolete and then go back to precedent or specific nodes of the configuration and configuration type.

Practically, a reconfiguration plan is generated by a planner, and it expresses how to transit from a departure configuration node (source architecture) to an arrival configuration node (target architecture). The generated plan can then be transmitted to the executor of the autonomic manager for automatic deployment. More details on how to automate the generation and deployment of a reconfiguration plan can be found in Section 3.4.

3.3.2 Illustrative scenarios

Considering using SemVer with version numbering of X.Y.Z, we put in action the evolution graph composed of configuration type nodes and configuration nodes into three scenarios: impactless patch changes, i.e. Z-changes (Section 3.3.2.1), compatible minor changes, i.e. Y-changes (Section 3.3.2.2), and incompatible major changes, i.e. X-changes (Section 3.3.2.3). We give an example of each one taken from the GDE case study to illustrate how each category of version changes can be performed using our evolution graph.

In order to apply a version change, we propose that architects act as follows, at first from the type view and then from the instance view:

S1) In the type view, create and commit a new configuration type node.

S2) In the instance view, create and commit a new configuration node that conforms to a configuration type by choosing which instances are replaced for executing the new version, and which instances keep executing the previous version.

S3) In the case where a problem occurs when executing the reconfiguration plan, it is possible to return to the previous configuration.

Importantly and as per usual, note that configuration type nodes and configuration nodes that are already committed in the evolution graph can not be modified any more, and that, before being deleted, a microservice type should be considered "deprecated" during at least one configuration type.

As presented in **S**2), when creating and committing a new configuration node, a choice must be made of which instances are being replaced and will execute the new

version, and of which instances keep executing the previous version. We define a percentage (`x`) of instances of the previous version that will be replaced for executing the new version in the next configuration, so that we have three main strategies:

— Zero-replacement/addition strategy ($\mathtt{x} = 0$): We do not remove any instances from the previous version, and the new version and the previous version are co-existing. The new instances will execute the new version, and we change nothing for the instances that execute the previous version.

— Partial-replacement/replace strategy ($\mathtt{x} \in]0, 1[$): The new version and the previous version are co-existing, but only `x` percent of instances of the previous version will be replaced for executing the new version in the next configuration.

— Total-replacement/removal strategy ($\mathtt{x} = 1$): all instances of the previous version are moved to executing the new version in the next configuration.

For each strategy, the actions that should be performed in the next configuration to commit are different. The strategy is used to build the new configuration from the previous one. Let define $m \geq 0$ as the number of instances of the previous version and $n \geq 0$ the number of instances of the new version. Note that whatever strategy is chosen, the number of instances of the new version does not necessarily have to be less than the number of instances of the previous version.

Before the development of the strategies, recall that, for the client-server pattern, we link/unlink a client to/from a server, and for the publish-subscribe pattern, we link/unlink a producer to/from a broker and link/unlink a consumer to/from a broker, where a broker is a publish-subscribe connector and the relationship between a producer and a consumer is "many to many". We now list for each strategy the actions that can be followed to build the next configuration in our evolution graph:

— Zero-replacement/addition strategy:

1) Create n new version microservice instances.
2) For each new version instance: link client and server, link producer and consumer to publish-subscribe connector.

— Partial-replacement/replace strategy:

1) Create n new version microservice instances.
2) For each new version instance: link client to server, link producer and consumer to publish-subscribe connector.
3) Calculate and choose $m \times \mathtt{x}$ microservice instances of the previous version to be replaced for executing the new version.

4) For each instance to be replaced, unlink client from server, unlink producer and consumer from publish-subscribe connector.
5) Remove the instances of the previous version.

— Total-replacement/removal strategy:
1) Create n new version microservice instances.
2) For each instance of the new version, link client to server, link producer and consumer to publish-subscribe connector.
3) For all m microservice instances of the previous version, unlink client from server, unlink producer and consumer from publish-subscribe connector.
4) Remove these m instances of the previous version.

Then, in the next sections, we illustrate with examples from the GDE case study the possible actions that can be taken for each category of version changes.

3.3.2.1 Patch change

Patch changes are for example due to correcting bugs or improving implementation, and have no impact on other elements. It is the simplest scenario because only microservice types and microservice instances will change, no other elements like contracts or connectors need to change. With the case study of Figure 3.1, let us suppose that a new version, e.g. `1.0.1`, of the `authentication_service` microservice comes out. The previous version `1.0.0` will continue to be supported and may still be instantiated.

We list below the detailed steps to follow with this "patch" scenario:

1) Create a new microservice type `Auth_1_0_1`, of which `Auth_1_0_1.identifier = authentication_service` and `Auth_1_0_1.version = 1.0.1`.
2) Add this new microservice type `Auth_1_0_1` into the next configuration type `CT`.
3) Check the `CT.isInstantiable()` property, and if so, commit this configuration type to the evolution graph, so that a new node `CT` is available.
4) Create a new configuration `C`, and set the new committed configuration type `CT` as its reference type, i.e. `C.type = CT`.
5) Choose a strategy of which instances are being replaced and will execute the new version (the value of `x`). All of three strategies are valid.
6) Perform the actions of the chosen strategy to build the new configuration `C`. For example, if we choose "zero-replacement strategy", then we create one microservice instance of new version, i.e. `as_1` and `as_1.type = Auth_1_0_1`, and we link this

new instance `as_1` by a client-server connector to another instance `us_1` that is also linked to the previous version instance.

7) Check the `C.isDeployable()` property, and if so, commit this configuration to the evolution graph, so that a new node `C` is available for being a configuration for the reconfiguration planner.

3.3.2.2 Minor change

Minor changes are backward-compatible changes. It becomes more complex because other elements, such as contracts or connectors, also have to change. In the case study of Figure 3.1, we imagine a simple concrete example illustrated by an object diagram in Figure 3.10 (the orange objects in the diagram are of the new version). In the initial type model, we have two microservice types, `AuthT1` and `UserT1`, that are connected by a client-server connector type in the configuration type. A microservice type `AuthT1` consumes a client contract type `cct1` provided by another microservice type `UserT1` through the corresponding server contract type `sct1`. Initially, all elements are in version `1.0.0`.

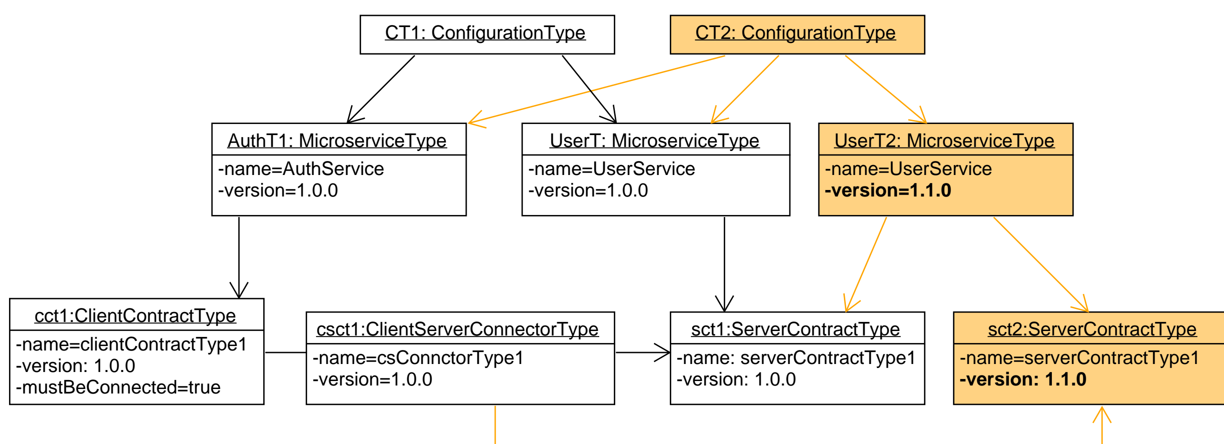


Figure 3.10 – Example of object diagram of type model with a possible minor change

Then, we develop the scenario of a minor change from this initial configuration type `CT1` to a new configuration type `CT2`: server contract type `sct1` is changed compatibly from version `1.0.0` to `1.1.0`, which introduces a new server contract type `sct2`. This new server contract type may have no connector attached. A possible example of minor changes in server contract type are adding a new operation declaration in this contract type. We list the detailed steps to operate this “minor” change:

1) Create a new microservice type `UserT2`, of which `UserT2.identifier = user_service` and `UserT2.version = 1.1.0`.
2) Create a new server contract type `sct2` for microservice type `UserT2` with `sct2.identifier = serverContractType1` and `sct2.version = 1.1.0`.
3) Add the new microservice type `UserT2` and the new client contract type `sct2` into the next configuration type `CT2`.
4) Check the `CT2.isInstantiable()` property, and if so, commit this configuration type into the evolution graph so that a new node `CT2` is available.
5) Create a new configuration `C` with `C.type = CT2`;
6) Choose and apply the strategy of which instances are being replaced and will execute the new version (the value of `x`). All of the three strategies are valid.
7) Perform actions of the chosen strategy to build the new configuration `C`. For example, if we choose "zero-replacement strategy", then we create one microservice instance of the new version, i.e. `as` and `as.type = AuthT2`, and we link this new instance `as` to another instance `us, us.type = UserT1` that is also linked to the instance of the previous version.
8) Check the `C.isDeployable()` property, and if so, commit this configuration to the evolution graph so that the new node `C` is available for being a configuration for the reconfiguration planner.

3.3.2.3 Major change

Major changes are changes that are incompatible. This type of changes can cause a series of changes in other elements, including contracts and connectors. Usually, major changes will lead to the changes of other microservices. We imagine a simple example illustrated by the object diagram in Figure 3.11. We use the same initial type model as the example in previous section of minor changes. Then, we suppose a scenario of a major change from this initial configuration type `CT1` to a new configuration type `CT2`: client contract type `cct1` is changed incompatibly from version `1.0.0` to `2.0.0`. This also leads to a major version change for microservice types, connected server contract type, and the connector type. A possible example of a major change in a client contract type is modifying an operation declaration in the client contract type so that it is no more compatible with the linked server contract type. We list the steps to proceed with this "major" scenario:

1) Create new microservice types `AuthT2` and `UserT2`, of which `AuthT2.identifier = authentication_service` and `AuthT2.version = 2.0.0`, and `UserT2.identifier = user_service` and `UserT2.version = 2.0.0`.

2) create a new client contract type `cct2` for microservice type `AuthT2` with `cct2.identifier = clientContractType1` and `cct2.version = 2.0.0`.

3) Create the corresponding server contract type `sct2` for `UserT2` with `sct2.identifier = serverContractType1` and `sct2.version = 2.0.0`, and satisfying `cct2.isCompatible(sct1) = false` $\wedge$ `cct2.isCompatible(sct2) = true`.

4) Add `AuthT2`, `UserT2`, `cct2`, and `sct2` into the configuration type `CT2`.

5) Check `CT2.isInstantiable()` before committing.

6) Create a new configuration `C` with `C.type = CT2`.

7) Choose and apply the strategy of which instances are being replaced and will execute the new version (the value of `x`), all of three strategies are valid.

8) Perform the actions to build the new configuration `C`. For example, if we choose "zero-replacement strategy", then we create `as` with `as.type = AuthT2`, `us` with `us.type = UserT2`, and we link `as` with `us`;

9) check `C.isDeployable()` befor committing.

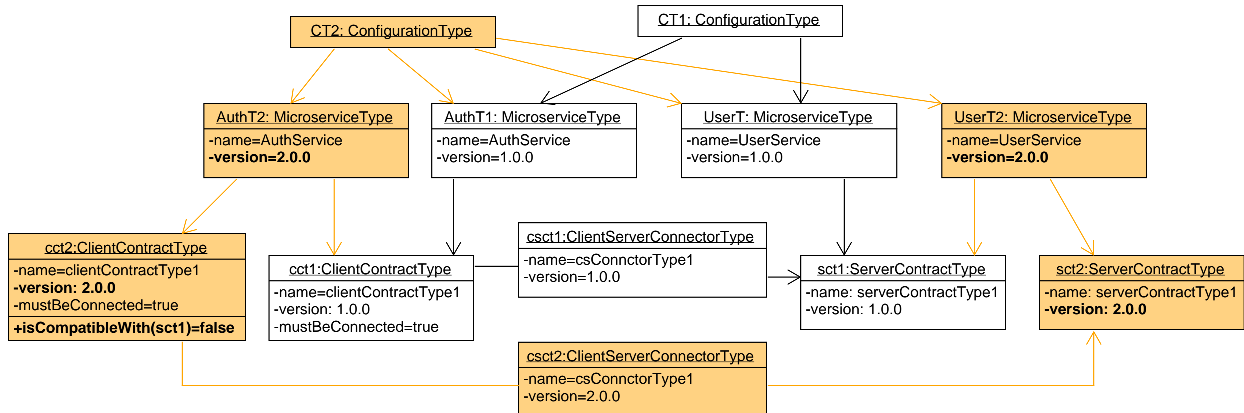


Figure 3.11 – Example of object diagram of type model with a possible major change

3.4 Implementation in MIMOSAE

To validate the feasibility of our runtime models and evolution graph, we engineer our solutions in a prototype. We implement the MAPE-K control loop, but in a semi-automatic way, that is, we manually provide the necessary information about the version change, such as a source configuration node and a target configuration node, so that the planning and execution of the evolution can then be performed in an automatic manner.

The implementation of the prototype, named MIMOSAE (for MIcroservices MOdel for verSion mAnagement with Evolution graph), can be found at the following URL: https://gitlabev.imtbs-tsp.eu/mimosae/mimosae [Mim, 2022]. Technically, in this prototype implementation, the planning is performed using a PDDL AI planner [Haslum et al., 2019], and the executor uses the Kubernetes API [1] and the Docker API [2] to deploy and reconfigure the managed application. We describe below the planner and the executor.

3.4.1 PDDL planner

In this section, we first introduce what is the Planning Domain Definition Language and some of the PDDL concepts. Then, we give a general domain description of our proposed runtime models of microservice-based applications. We demonstrate next how to use PDDL problem description files to define the scenarios of version changes from one configuration to another in our GDE case study. Finally, we show the results with the obtained reconfiguration plan generated by the PDDL planner with the above domain file and problem file. The full version of the domain file and problem files can be found in Appendix A.

Introduction to PDDL

PDDL is designed to solve automated planning and scheduling [Haslum et al., 2019], a branch of Artificial Intelligence (AI). As a formal representation language to express general planning tasks and models, it can also be used for architectural evolution path generation problems [Barnes et al., 2013, Méhus, J.-E. and Batista, T. and Buisson, J., 2012].

PDDL specifies what needs to be done for a planning problem rather than how to do it. As an entry to the PDDL solver, a planning task is organised according to a list of *objects* under consideration and *predicates* that are the properties of objects. It is then described by an *initial state* as a starter, a *goal* to achieve, and a list of *actions* that can be executed. Each action has *pre-conditions* that define the constraints on the states before an action can be performed, and *post-conditions*, also called *effects*, that are the state changes after the execution of the action. The result of the PDDL solver is an ordered list of sets of actions, which can be performed in parallel.

A PDDL specification declares planning problems in two separated files:

(1) a *domain* file that defines a general model of behaviours and operators for a specific application, including predicates and possible actions.

1. The Kubernetes API: https://kubernetes.io/docs/concepts/overview/kubernetes-api/
2. Docker Engine API: https://docs.docker.com/engine/api/

(2) a *problem* file that defines a specific problem instance in the domain, including specific objects, initial state, and goal condition.

PDDL domain description for microservices

In MIMOSAE, the architectural entities such as microservices, contracts, connectors are encoded as objects. The relationships between these entities are defined as predicates, which are facts that can be true or false. Each reconfiguration operation, such as create and remove microservices, or link and unlink microservices, is described into a PDDL action explaining the preconditions and effects in terms of the above predicates.

We only display in Listing 3.1 part of the definitions in the domain file in order to explain the main idea, the full version being available in Appendix A.1.

A domain description file always starts by the declaration of its name preceded by the `domain` keyword: `microservice_planner_domain`. Then, the requirements of this domain are defined: usage of types for objects and some ADL features, i.e. disjunctions and quantifiers in preconditions and goals, and quantified and conditional effects.

Listing 3.1 Example of the PDDL domain file for MIMOSAE

```
1  (define (domain microservice_planner_domain)
2    (:requirements :adl :typing)
3    (:types
4      architecturalentity - object
5      microservice - architecturalentity
6      pubsubconnector - connector
7      ...
8    )
9    (:predicates
10     (microservice ?m - microservice)
11     (client_server_link ?mc - microservice ?ms - microservice)
12     ...
13   )
14   (:action create_microservice
15     :parameters (?m - microservice)
16     :precondition (and (not (microservice ?m)))
17     :effect (and (microservice ?m))
18   )
19   ...
20 )
```

The primitive types of a domain definition, identified by the `types` keyword, represent objects existing in the planning problem. Similar to classes and subclasses in object-oriented programming, it is possible to derive the necessary types, for instance, `microservice` and `architecturalentity`.

The `predicates` section contains state binary variables that denote facts about objects: e.g., the predicate `(microservice ?m)` to state that the microservice m already exists, `client_server_link` to state that there exists a client-server link between two

microservices.

Actions define operators to transition between states. Each action has parameters, preconditions that define the state when the action is applicable, and effects that constrain the state after the action is applied. For example, action `(create_microservice)` is applicable only when `(microservice ?m)` is false and specifies that `(microservice ?m)` becomes true.

PDDL problem description for GDE scenarios

We demonstrate a complete scenario of a reconfiguration in the GDE system due to a version change. This scenario focuses on the transition from one source configuration to a target configuration, without considering when this transition can be performed safely (this is the concern of Chapter 4). The process of this scenario consists of several steps:

S1) Create an empty initial configuration that contains five microservices, all in version `1.0.0`: `Authentication Service`, `User Service`, `Project Service`, `File Service`.

S2) Change from the initial configuration created in the first step to a new configuration. The changes include: replace microservice `File Service` with a minor version, from `1.0.0` to `1.1.0`, create a new microservice `Logger Service` in version `1.0.0`, and connect the new version of `File Service` with `Logger Service` via a publish-subscribe connector. This process is shown in Figure 3.12.

S3) Transition from the current configuration to an empty configuration that ends the distributed application.

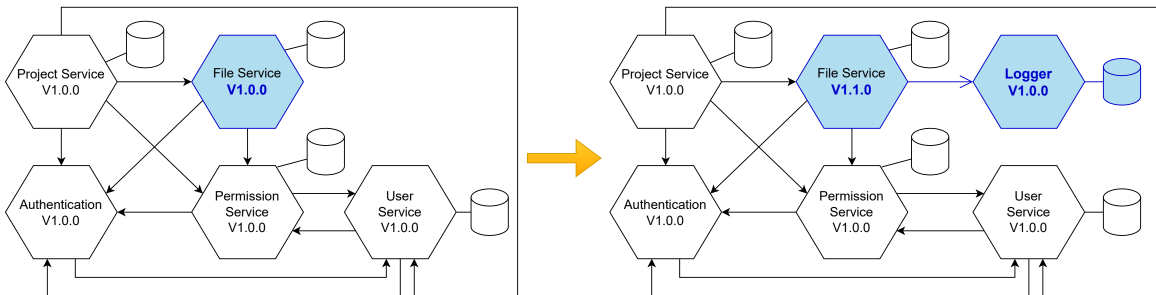


Figure 3.12 – Scenario implemented in MIMOSAE (described in step **S**2)

In our implementation, we generate the problem description file programmatically from runtime models. An example of part of the programmed problem file for **S**2) is shown in Listing 3.2, the full version being available in Appendix A.1.

The problem description, like the domain description, is named by a `problem` keyword: `microservice_planner_problem`. It includes a reference to the associated domain

Listing 3.2 Example of the PDDL problem file for the scenario of minor version change

```
1  (define (problem microservice_planner_problem)
2    (:domain microservice_planner_domain)
3    (:objects
4      fsi_fs_1_0_0 - microservice
5      fsi_ps_1_1_0 - microservice
6      pubsubci_pubsubconnector_1_0_0 - pubsubconnector
7      ...
8    )
9    (:init
10     (microservice fsi_fs_1_0_0)
11   )
12   (:goal (and
13      (microservice fsi_fs_1_1_0)
14      (pubsubconnector pubsubci_pubsubconnector_1_0_0)
15      (producer_to_connector_link fsi_fs_1_1_0 pubsubci_pubsubconnector_1_0_0)
16      ...
17   )
18   ...
19 )
```

file. The body part of the problem description starts by listing objects that will be used in the initial state and in the goal: e.g. `fsi_fs_1_0_0` and `fsi_ps_1_1_0`, and `pubsubci_pubsubconnector_1_0_0`. Then, the `init` section defines the initial state of the problem instance by a list of facts, and the `goal` section defines a condition that should be satisfied at the end of a valid plan: e.g., the initial state `fsi_fs_1_0_0` that is an instance of `File Service` in version `1.0.0`, and the goal contains `File Service` in new version `1.1.0` and linked it to a publish-subscribe connector.

Reconfiguration plan generated by PDDL planner

The solution to a planning problem is a plan, and a plan is a sequence of sets of parallel actions. Listing 3.3 shows an example of a generated reconfiguration plan. It is the part that presents how the scenario of Figure 3.12 will change step by step from a configuration (identified by `d543944ce9`...) to another configuration (identified by `cddf8368b8`...). The actions in one "in parallel" section can be executed concurrently, and the sequence of "in parallel" sections is executed sequentially.

3.4.2 Executor

The plan generated by the planner is automatically processed by an executor to perform the reconfiguration and redeployment. Technically, we implement the executor by using Docker and Kubernetes APIs. Docker is a tool that allows the containerization of one or multiple services, and Kubernetes is a container orchestration tool. The main steps of our executor are executed as follows (the communication diagram of the

Listing 3.3 Example of the plan generated by the PDDL planner

```
1   From d543944ce99c8c0760c1b28b554cd6 ... to cddf8368b811326a693f0b0bbc400c ...
2   in parallel:
3     [action=create_pubsubconnector, args=[pubsubci_pubsubconnector_1_0_0]]
4     ...
5   in parallel:
6     [action=unlink_client_server, args=[fsi_fs_1_0_0, ai_auth_1_0_0]]
7     [action=disconnect_microservice_from_databasesystem, args=[fsi_fs_1_0_0,
        dbfsi_dbfs_1_0_0]]
8     [action=unlink_client_server, args=[fsi_fs_1_0_0, usi_us_1_0_0]]
9     [action=unlink_client_server, args=[fsi_fs_1_0_0, permsi_perms_1_0_0]]
10    [action=unlink_client_server, args=[psi_ps_1_0_0, fsi_fs_1_0_0]]
11    [action=create_microservice, args=[fsi_fs_1_1_0]]
12    [action=create_microservice, args=[logsi_ls_1_0_0]]
13    ...
14  in parallel:
15    [action=remove_microservice, args=[fsi_fs_1_0_0]]
16    [action=link_client_server, args=[psi_ps_1_0_0, fsi_fs_1_1_0]]
17    [action=link_client_server, args=[fsi_fs_1_1_0, ai_auth_1_0_0]]
18    [action=link_client_server, args=[fsi_fs_1_1_0, usi_us_1_0_0]]
19    [action=link_client_server, args=[fsi_fs_1_1_0, permsi_perms_1_0_0]]
20    [action=connect_microservice_to_databasesystem, args=[fsi_fs_1_1_0,
        dbfsi_dbfs_1_0_0]]
21    [action=link_producer_to_pubsubconnector, args=[fsi_fs_1_1_0,
        pubsubci_pubsubconnector_1_0_0]]
22    [action=link_consumer_to_pubsubconnector, args=[logsi_ls_1_0_0,
        pubsubci_pubsubconnector_1_0_0]]
23    ...
```

implemented executor can be found in Appendix A.2):

1) Receive a reconfiguration plan from the planner via the executor front-end.
2) Validate the correctness of the plan (this is a kind of “defensive programming”).
3) Create in parallel the actions extracted from the plan.
4) For each action, look for the configuration files from the codebase of the project (Kubernetes `.yaml` configuration files and `Dockerfiles`).
5) Build Docker images by using Docker APIs.
6) Parse Kubernetes `.yaml` configuration files to create different Kubernetes entities and to initiate an internal graph of dependencies between these Kubernetes entities.
7) Call Kubernetes APIs to deploy all entities respecting dependencies.

3.5 Discussion

As detailed in the previous sections, our runtime model separates types from instances and considers both client-server and publish-subscribe (either channel-based or topic-based publish/subscribe systems) communication modes. In addition, the evolution graph tracks the evolution trajectory of the microservice architecture.

Some recent works have considered multi-version microservice management with contributions complementary to ours. The authors of [Sampaio et al., 2017] propose to build a microservice evolution model by observing the managed system under execution. The model combines type and instance elements, i.e. with no separation of concerns, which makes it difficult to trace software evolution. In addition, the model includes versioning of only some of the model elements, namely, application, service, and operation. Similarly, the model in [Sorgalla et al., 2018] is derived in a bottom-up approach from the concepts of a specific ecosystem of microservice technologies. The two works only focus on synchronous interactions between microservices, thus, for instance, ignoring the concept of connector (because connectors are not mandatory in client-server connections). By contrast, our proposal separates the type model from the instance model and exposes the validity of the two models (resp. the "instantiable" and "deployable" properties), and the conformity between the two models. Then, all the elements of the type model are versioned. In addition, we include database systems and asynchronous communication.

The authors of [Ma et al., 2019] track microservice dependencies and versions by analysing code (JAVA annotations) and runtime entities (JAVA reflection mechanism). They are capable to detect dependency errors at runtime. The dependency graph is computed through a series of chain searches, and version management is based on chain manipulations. Communications can be synchronous or asynchronous, but only in the context of the Spring Cloud ecosystem. In our work, the dependency graph is included in the model so that errors are detected at design time when committing a configuration, and version management is based on the manipulation of model elements and is logged.

In all the previous works, graphs and models represent microservice architectures at a given instant. In our proposal, we record the trajectory and history of how the system evolves over time. Each node in our evolution graph is a snapshot of the running system configuration as version changes are applied.

The authors of [He et al., 2020] propose a formal model of version dependency for multi-version coexisting microservice systems. The model is generated from configuration files and is used in a greedy-based optimisation algorithm to generate an optimal deployment plan. The authors of [Rajagopalan and Jamjoom, 2015] propose a tool to record microservice dependencies at every version update by taking an OS-like package management approach. They create a version timeline per microservice that includes version dependencies to record revision histories, e.g. major or minor updates and dependency requirements about compatible versions with other microservices. In our work, we present microservice relationships in a more detailed manner, and we

trace evolution histories at the granularity of a configuration, which includes a set of microservices in applications, not just at a single microservice.

The authors of [Tao, 2019, Boyer et al., 2018] propose a data structure modelling the architecture of a microservice-based application, which specifies how microservices are deployed on PaaS sites and how they are configured with PaaS-common and PaaS-specific configuration attributes. The main goal of their architectural model is to represent microservices running on heterogeneous PaaS platforms in the context of the Cloud, which context is different from ours [3]. To manipulate microservices, their solution relies on the operations provided by PaaS sites. In our work, our runtime models and evolution graphs do not focus on the infrastructures for deploying microservices, but rather on the microservices.

Besides model-at-runtime, another way to model software architecture is Architecture Description Language (ADL). The work of [Huynh, 2017] applies this approach to model software architectures and their variability. However, this work contributes to another related research domain: software product lines (SPLs), which is out of the scope of this thesis.

3.6 Conclusion

In this chapter, we have presented our contributions to the runtime model and evolution graphs for microservices and their version management. Our modelling separates the view of types from the view of instances, adds version management to every model element, and considers both synchronous and asynchronous communication modes. Our model is reified at runtime to be part of the Knowledge base of the MAPE-K control loop. Managed elements are then mirrored in the runtime model. With the evolution of the microservice application, if there are changes occurring in the application, the models will also change, first the type model and then instance model, and vice versa. [4]

Our evolution graph is used to track the evolution trajectory of the microservice architecture. Every time the artefacts of a microservice type is added or removed from the implementation repository, the type model can evolve to take into account the evolution, and a new type node is created and committed in the evolution graph. Validation is only possible if the microservice types are instantiable. The new node represents the set of software artefacts (microservices, connectors, contracts, etc.) that

3. In Chapter 4 about our second contribution, we describe the distributed application model we focus on: a distributed application is contained in one and only one distributed component (e.g. Clouds), the other distributed components acting as actors for this distributed application

4. To obtain a fully causal relationship, the model should change when the managed elements change, e.g. in case of failure. But, this part is out of the scope of this thesis.

can be used for building the managed system. The second part of the evolution graph corresponds to the snapshots of deployed configurations. Validation is only possible if the microservice instances are deployable. Every time a decision is made by software architects, a new instance node is created and committed in the evolution graph. Such a node represents the set of deployed entities (deployed instances of microservices, connectors, etc.) when the reconfiguration plan is computed and executed to effectively change the distributed application.

Finally, our solution is implemented in the prototype named MIMOSAE (for "MIcroservices MOdel for verSion mAnagement with Evolution graph"). It can be found at the following URL: https://gitlabev.imtbs-tsp.eu/mimosae/mimosae. When a new configuration to deploy is committed in the evolution graph, an AI Planner can compute a plan of deployment actions to transition the system towards this new configuration. This plan is automatically executed to perform the deployment.

In the next chapter, we will present our second contribution about when and how the reconfiguration for updating microservices can be performed consistently while client service calls keep arriving. This problem is named dynamic software updating.

Chapter 4

Snapshot-based Dynamic Software Updating of Microservices

Contents

4.1 DSU Problem **66**

4.1.1 GDE use case 66

4.1.2 Role of the DSU algorithm 66

4.1.3 Update conditions 68

4.2 Distributed System and Application Models **69**

4.2.1 Distributed system model 71

4.2.2 Distributed application model 72

4.3 Snapshot-Based Update Setting **74**

4.3.1 Microservice execution model 74

4.3.1.1 Path of application messages 74

4.3.1.2 Link dependencies 75

4.3.1.3 Message type dependencies 76

4.3.2 Adding continuity of service 80

4.3.3 Essential and non-essential changes 81

4.3.4 Correct dynamic update 83

4.3.5 Snapshot-based definitions of the update conditions 84

4.3.5.1 Quiescence 84

4.3.5.2 Freeness 85

4.3.5.3 Essential freeness 85

4.4 Snapshot-Based DSU Algorithm **86**

4.4.1 Termination detection of collaborations 86

4.4.2 DSU algorithm and updating strategies 88

4.4.2.1 Overview of the DSU algorithm 89

4.4.2.2 DSU algorithm for quiescence 89

4.4.2.3 DSU algorithms for essential freeness and freeness 90
4.5 Discussion **92**
4.6 Conclusion **94**

This chapter addresses the issue of dynamic software updating (DSU) for microservice-based applications, a process for safely modifying a running system in place without stopping it, especially for long-time running or frequently-executed activities [Seifzadeh et al., 2013]. The DSU problem can be decomposed into the following two sub-problems to be solved in order:

(i) Model the distributed application and version changes, and express the update condition in this model in such a way that the update does not lead to semantic inconsistencies: e.g. which architectural elements of the distributed application evolve and which must evolve together, what is a client service call, and can microservices be updated in the middle of a client service call?

(ii) Specify the DSU algorithm that monitors the running system for the update condition and that performs the update using a given strategy: e.g. does the system allow both versions of a microservice to run simultaneously, should the DSU algorithm block certain messages, or certain architectural elements?

In Section 4.1, we provide an overview of the DSU problem, including an illustrative case study, the role of the DSU algorithm in distributed applications, and a reminder of the meaning of the three safe update conditions that will be reformulated using snapshots in this chapter.

In Section 4.2, we introduce the distributed system model and the distributed application model. The former model presents the assumptions of the distributed system and introduces the concept of consistent distributed snapshot. The latter model describes common elements of distributed applications, including architectural elements such as the front end and an abstraction of the algorithm of microservices.

In Section 4.3, we complement the models of types and instances of Chapter 3 with an execution model of microservices to be capable of answering the following two questions: (i) what are the messages to be sent in the future of an in-progress client service call? (ii) is a microservice change an essential one or not? We then formulate the properties of continuity of service, essential change, correct dynamic update, and we give snapshot-based definitions of the update conditions of the literature.

In Section 4.4, we present our snapshot-based DSU algorithm that contains the detection of termination of collaborations and the maintenance of update conditions with different update strategies.

4.1 DSU Problem

The goal of this section is to provide background information before detailing our solutions about the DSU problem. We begin with detailing a use case of GDE case study in Section 4.1.1 to help us illustrate more clearly the proposed concepts and algorithms. Then, in Section 4.1.2, we outline the DSU algorithm for microservice-based applications to give a global overview. Finally, in Section 4.1.3, we informally remind the three update conditions defined in previous research, but this time illustrated with our use case.

4.1.1 GDE use case

Throughout the Sections 4.2 to 4.4, in order to better explain our proposals, we use the illustrative use case “attach a file to a project” of the GDE case study. The GDE use case has been introduced in Chapter 3.1 for its global structure; it is a system to manage scientific research data. It contains six microservices. We imagine a scenario of attaching a simulation file to a specific project. To service the `AttachFileToProject` request from an actor, the authentication and authorization services should be firstly verified respectively through `Authentication Service` and `Permission Service`. Then, it should check whether the project and associated files already exist in the system; this is done by calling `Project Service` and `File Service`. If everything goes well, `Project Service` performs the action and returns a result to the actor.

Figure 4.1 presents the UML sequence diagram with the message exchanges, with some interesting local states and with coloured portions of lifelines that we will refer to throughout this chapter, and more particularly in Section 4.2.1. The design of the use case “attach a file to a project” is asynchronous in this sequence diagram: messages are events, and for the sake of simplicity, we ignore the broker (assuming that there is only one broker in the architecture). In other words, the diagram depicts the exchange of events between microservices and every event is published to the broker, which forwards the event to the interested microservice.

4.1.2 Role of the DSU algorithm

Conceptually, the role of the DSU algorithm is to move a microservice-based application from a source configuration to a target configuration without shutting down the system, e.g. from configuration $\mathcal{C}$ to $\mathcal{C}'$ in Figure 4.2. The DSU algorithm is executed by the autonomic manager [Kephart and Chess, 2003]. It involves distributed algorithms whose role is to detect the update condition and to reconfigure the distributed

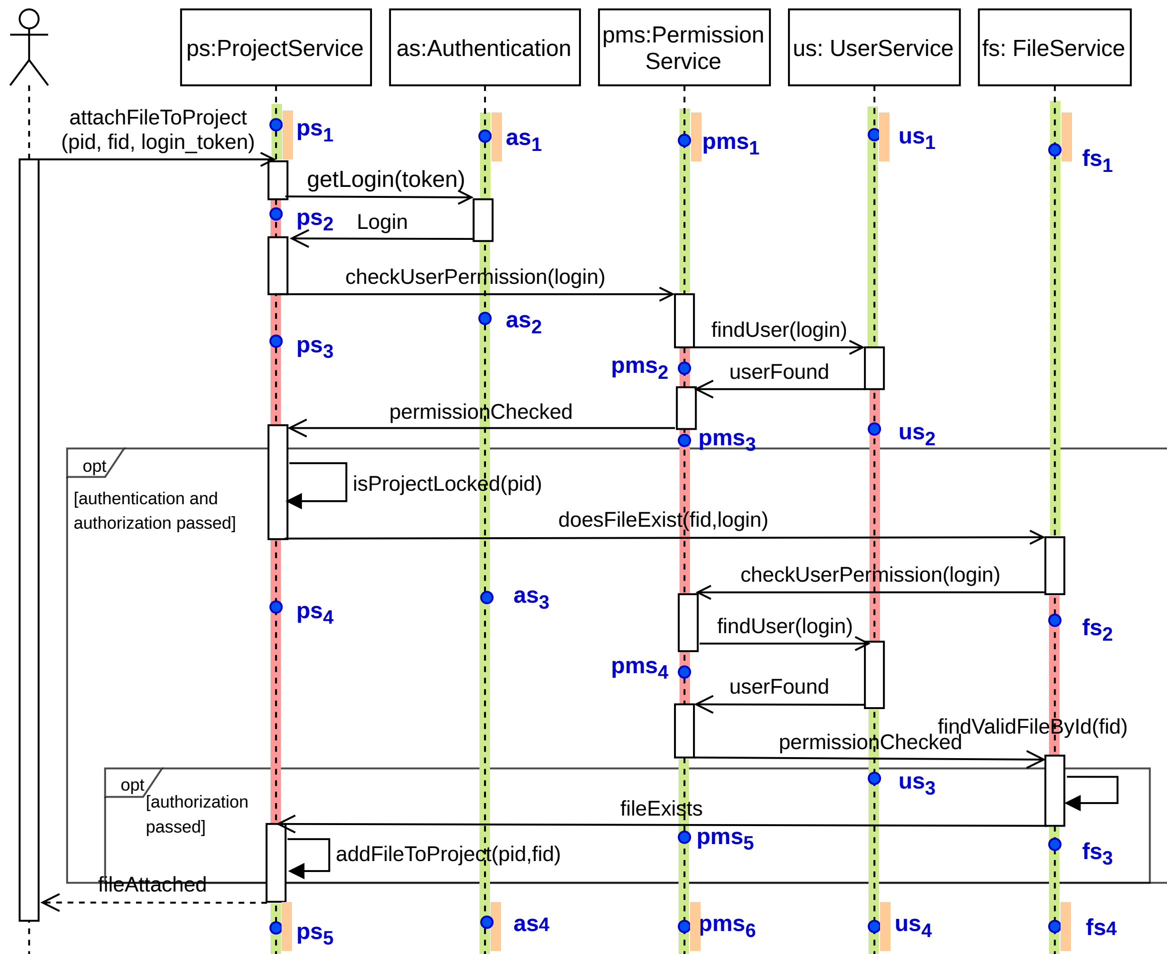


Figure 4.1 – Sequence diagram of the use case `AttachFileToProject`—publish-subscribe communication (blue points: local states of microservices; coloured portions of lifelines: safe periods for updating microservices according to different update conditions)

application using primitives such as create/remove microservices and link/unlink microservices. These distributed algorithms are not constrained by links between microservices, i.e. they may not use microservice links and the autonomic manager has bidirectional links with all microservices.

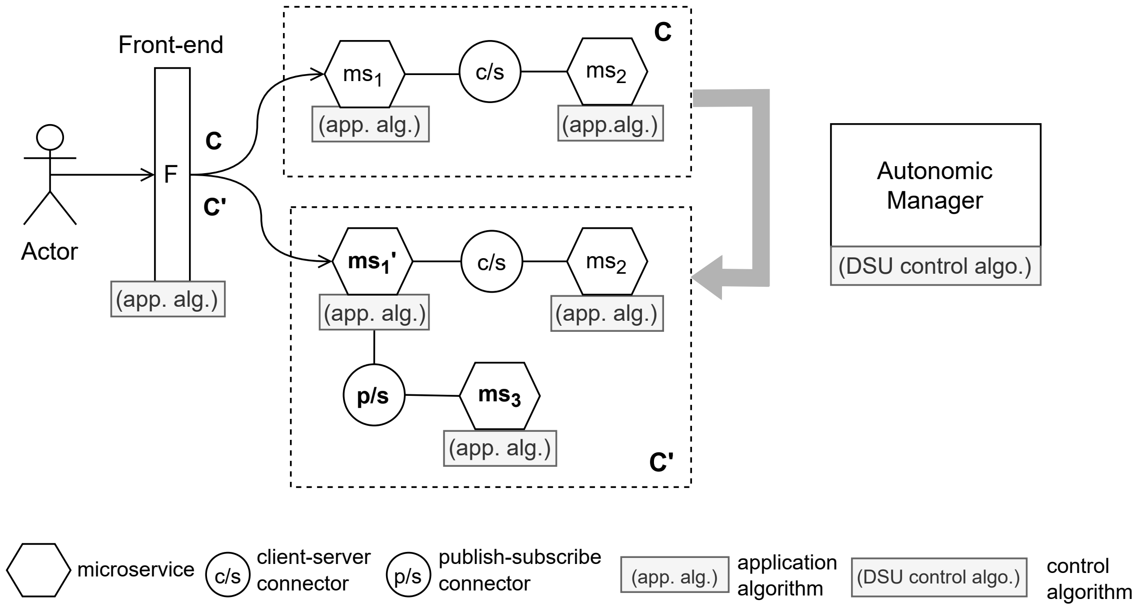


Figure 4.2 – Global view of DSU algorithm for microservices

There are two categories of messages: application messages are due to application actions, namely the application or microservice algorithm ("app. algo." in Figure 4.2), and control messages are sent by the DSU algorithm, namely the control algorithm ("DSU control algo."). The application messages are sent either through client-server connectors ("c/s") or publish-subscribe connectors ("p/s"). To achieve continuity of service during updates, clients access the application through a front-end. The role of the front-end is to redirect messages to the microservice ms_1 in configuration $\mathcal{C}$ to its replacement ms'_1 in the next configuration $\mathcal{C}'$. For the sake of simplicity, we consider that there is only one front-end, denoted by F, for the distributed application.

4.1.3 Update conditions

Update conditions, which respond to the question of when updating can be performed safely, have been the focus of previous research. There are four update conditions that have been introduced in Section 2.3.4. What we are interested in are the three conditions that are safe, namely quiescence, freeness and essential freeness. As a reminder, we present again these three update conditions briefly and illustrate them in Figure 4.1 with our use case "attach a file to a project".

The authors of [Kramer and Magee, 1990] define quiescence as the property that must be true of a component state such that a version change is permitted and is guaranteed to be safe: the component must not be serving transactions or initiate new transactions, and neither must every component that is directly nor indirectly capable of initiating transactions on this component. For example, in Figure 4.1, participant pms is quiescent at global state $(ps_1, as_1, pms_1, us_1, fs_1)$ and at global state $(ps_5, as_4, pms_6, us_4, fs_4)$, but not somewhere in between. Quiescence corresponds to the orange bars in the figure.

In [Ma et al., 2011, Baresi et al., 2017], the authors propose version consistency as a global condition for safe updating and freeness as the corresponding local (component) condition. Considering a client service call, a component is free up to the receipt of the first transaction request and after servicing the last transaction request of the call. Thus, in Figure 4.1, participant pms is free at state $(ps_3, as_2, pms_1, us_1, fs_1)$ and at global state $(ps_4, as_2, pms_5, us_3, fs_2)$. Freeness corresponds to the green bars in the figure.

The authors of [Sokolowski et al., 2022] introduce the essential freeness property for systems based on both synchronous and asynchronous workflows. They distinguish between essential and non-essential changes. In case of a non-essential change, a component can be replaced as soon as it is not currently executing a task. In case of an essential change, the update condition is the same as for freeness. In Figure 4.1, if the change is non-essential, participant pms can be replaced at global state $(ps_3, as_2, pms_2, us_1, fs_1)$, which is between the first participation in the client service call and the second one. Essential freeness corresponds to the red bars in the figure.

Later in Section 4.3.5, these three update conditions will be formalised using the concept of consistent distributed snapshot, which is explained in the next section, and using our model of a distributed application.

By comparison, we depict in Figure 4.3 the update conditions in the sequence diagram of the same use case as of Figure 4.1, but using client-server synchronous communication. It demonstrates by an example why architects favour publish-subscribe communication: there exist more opportunities for updating in case of non-essential changes.

4.2 Distributed System and Application Models

In Sections 4.2.1 and 4.2.2, we define respectively the distributed system model, which, in a nutshell, is fault-free and asynchronous, and the distributed application model, which patterns the interactions around client-server and publish-subscribe

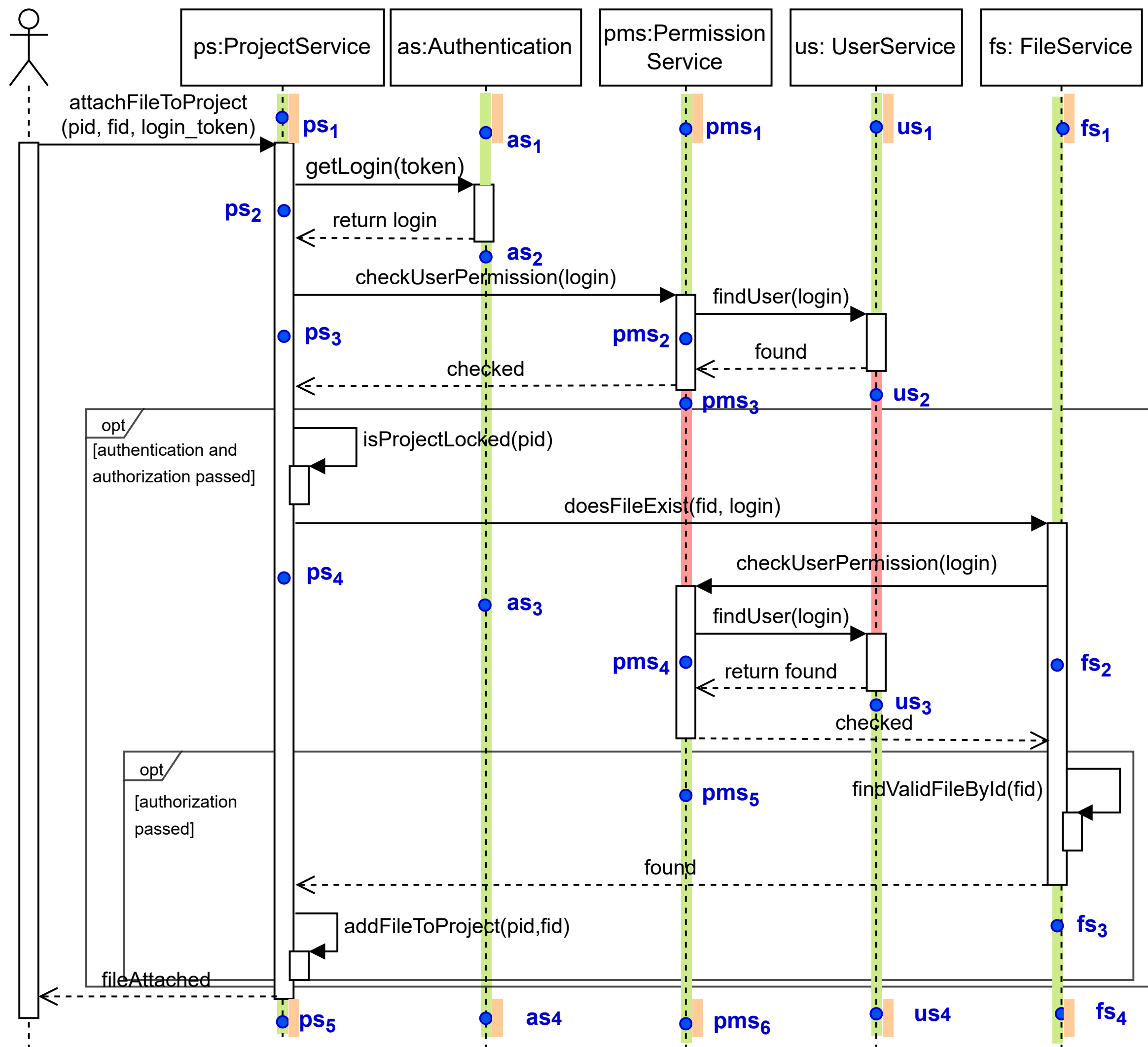


Figure 4.3 – Sequence diagram of use case `AttachFileToProject`—client-server communication (blue points: local states of microservices; coloured portions of lifelines: orange for quiescence, green for freeness, red for essential freeness in case of non-essential changes)

exchanges, and introduces the generic algorithm of microservices.

4.2.1 Distributed system model

The different parts of the distributed system may span multiple data centres, such that they constitute what we call distributed components. In this study, we focus on distributed applications that are contained in one and only one of these distributed components, and the other distributed components act as actors (in the UML sense of the term), the other actors being the end-users. Thus, the autonomic manager is not a bottleneck. In addition, we consider only fault-free distributed systems. Communication channels are reliable with no message creation, duplication, or alteration, and are such that they ensure that when a microservice sends a message to another entity, being another microservice or another architectural entity (front-end, publish-subscribe connector, or autonomic manager), the message is eventually delivered to the remote entity, and *vice-versa*. Finally, microservices and the other architectural entities are not subject to failures.

In order to simplify the presentation of the model without loss of generality, we assume the existence of a global clock that is not accessible by the architectural entities. We take the range $\mathcal{T}$ of the clock's ticks to be the set of natural numbers $\mathbb{N}$. Then, following [Lamport, 1978], the execution of a microservice is abstracted as a sequence $\sigma = a_0 a_1 a_2 \ldots$, a.k.a. trace, of atomic steps a_i, a.k.a. actions, that are either internal computations, reactions to the receipt of a message, or the emission of an application message. The actions of microservices are related by the *happened before* relation [Lamport, 1978] denoted by $\mapsto$.

In addition, since our solution is "snapshot-based", we introduce the definition of a consistent distributed snapshot. A distributed snapshot s locates a global state of the distributed application and is composed of a snapshot per microservice, say ms, with each microservice snapshot, denoted by $s[ms]$, that identifies the local state of the microservice ms. The set of microservices is denoted by S. According to [Chandy and Lamport, 1985], a distributed snapshot s is consistent if and only if there is no message seen as received but not sent in the global state, formally $consistent(s) = \forall ms_1 \in S, \nexists ms_2 \in S, s[ms_1] \mapsto s[ms_2]$. In other words, the recorded microservice and link states form a meaningful global state that may have existed. In Figure 4.1, distributed snapshots for the GDE case study are composed of microservice snapshots $(ps_i, as_j, pms_k, us_m, fs_n)$. For example, distributed snapshot $s = (ps_3, as_2, pms_3, us_1, fs_1)$ is inconsistent because it exhibits message `log(UserFound)` that is seen as received in $s[pms_3]$ without being seen as emitted in $s[us_1]$. Functionally speaking, an external

observer cannot evaluate the progress of the client service call "attach a file to a project" by considering distributed snapshot s: i.e. s has never existed in reality and corresponds to a "fake" photo of the distributed application. On the contrary, distributed snapshot $s = (ps_3, as_2, pms_3, us_2, fs_1)$ is consistent and can be used as a basis for evaluating for example update condition freeness.

The happened before relation on atomic steps is extended to consistent distributed snapshots as follows. Consistent distributed snapshot s_i happens before consistent distributed snapshot s_j if and only if s_j advances the execution of s_i in at least one microservice, formally $s_i \rightarrow s_j \;=\; consistent(s_i) \wedge consistent(s_j) \wedge \Big(\exists ms \in S, s_i[ms] \mapsto s_j[ms]\Big) \wedge \Big(\forall ms \in S, s_i[ms] = s_j[ms] \vee s_i[ms] \mapsto s_j[ms]\Big)$. Sequences of distributed snapshots that are related by the happened before relation are written by $s_i s_{i+1} s_{i+2}$.... For example, in Figure 4.1, $s_A = (ps_1, as_1, pms_1, us_1, fs_1) \rightarrow s_B = (ps_2, as_1, pms_1, us_1, fs_1)$ because both snapshots are consistent and $s_A[ps] \mapsto s_B[ps]$, the other local snapshots being equal. We also use intervals, for instance to specify that a distributed snapshot occurs between two others, e.g. $s \in]s_{begin}, s_{end}[$.

4.2.2 Distributed application model

As done in Chapter 3, the autonomic manager that is in charge of version management maintains two models of the distributed application: type and instance models. These two models are specified at design time and are available at runtime. It is important to note that when replacing a microservice, the correctness of the replacement is ensured by checking for type conformity. The deployed microservices constitute the configuration of the distributed application.

Since microservices are not subject to failures, the role of DSU is to transition the distributed application from a configuration $\mathcal{C}$ satisfying specification $\mathbb{S}_{\mathcal{C}}$ to a new configuration $\mathcal{C}'$ satisfying specification $\mathbb{S}_{\mathcal{C}'}$. We limit this study to updating due to version changes of microservices, and as done in [Ma et al., 2011] and in similar works, we assume that the same updating, but performed *offline*—that is, the system is shut down, updated, and restarted— is correct. Microservices may be stateful and have access to stable storage to enable state transfer: the state of a microservice is a set of values, i.e. conceptually a set of pairs `(name, value)`. However, as patterned in microservice-based architectures [Newman, 2015], in most cases, the stable storage system stays local to a microservice and is not centralised, e.g. every microservice may be attached to a local database.

As we said before, we have application messages manipulated by the application algorithm and control messages sent by the control algorithm. Since one of the roles of

the control algorithm is to observe the execution of the application algorithm, it must not influence the computation of the application algorithm. For the sake of brevity and without lack of generality, we ignore brokers and consider that published events are broadcast to consumers, i.e. the receiver of a published event is a microservice, even if it transits via a broker. In Figure 4.1, microservice *ps* publishes messages to microservice *as* and the intermediate broker is not depicted as a participant of the sequence diagram. Then, the application messages are the transaction requests (*req*) and replies (*rep*) of the client-server exchanges, and the published events (*evt*) of the publish-subscribe exchanges. Importantly, messages are sent following the FIFO order: FIFO order is easily implemented in request-response exchanges, for instance using TCP but cannot reasonably be assumed using topic-based brokers such as the ones implementing the AMQP or MQTT specifications. Therefore, following the end-to-end argument [Saltzer et al., 1984], we assume that FIFO order is implemented at the microservice level. Since we ignore communication failures, FIFO order is trivially implemented using, for instance, TCP links or sequence numbers, and for this reason, it is not shown in our algorithms.

For the termination detection of client service calls, and after [Tel, 2002], the set of states of a microservice is partitioned into two subsets, the active and passive states. Termination detection and updating are only allowed in passive states. More precisely, a microservice is active if an internal computation or the sending of an application message is applicable, and passive otherwise (e.g. the microservice is not executing an internal computation and is waiting for the receipt of an application message). In a passive state, only receipt actions are enabled, and microservices may send application messages to other microservices only when active. Initially, each microservice is active because it executes its initialization phase. For the sake of simplicity, we ignore the messages sent during the initialisation phase of the microservice.

Therefore, the application algorithm of microservice *ms* is abstracted in event-driven Algorithm 1, which consists of three actions:

— Action R_{ms} is enabled when an application message has arrived and *ms* is passive. It makes *ms* active and it is the receipt of an application message.

— The sending of an application message in Action S_{ms} is enabled only when *ms* is active, that is messages are not sent spontaneously but during an Action S_{ms} that is called by an Action R_{ms}.

— When the receipt of an application message finishes, Action I_{ms} is enabled and *ms* becomes passive, that is the next step of *ms* is the receipt of an application message.

Algorithm 1 Initial application algorithm — code at ms

1: Local variables:
2: $status \in \{active, passive\}$
3: R: {App. msg $\langle m \rangle$ has arrived at ms from $ms_s \wedge status = passive$}
4: receive $\langle m \rangle$ from ms_s
5: $status \leftarrow active$; treatment of $\langle m \rangle$, including calls of S
6: S: {App. msg $\langle m \rangle$ is being sent to $ms_d \wedge status = active$}
7: send $\langle m \rangle$ to ms_d
8: I: $\{status = active\}$
9: $status \leftarrow passive$

Algorithm 1 is going to be complemented for DSU (see Section 4.4.1).

4.3 Snapshot-Based Update Setting

In this section, we present a formal model for the updating of microservices in microservice-based architectures. On one hand, we complement the model of Chapter 3: the execution model of microservices (Section 4.3.1), the identification of their version changes (Section 4.3.3), and the definition of the correctness of a software dynamic update (Section 4.3.4). On the other hand, using the concept of consistent distributed snapshot, we formalise the properties to reach before updating a microservice: quiescence, freeness and essential-freeness (Section 4.3.5). Finally, since we consider version changes of a set of microservices (in a configuration), we can formulate the property of continuity of service (Section 4.3.2).

4.3.1 Microservice execution model

At design time, architects and developers build a model of the distributed application that is going to be used at runtime to detect the update condition. This model is decomposed into three parts: application messages including client-server messages and publish-subscribe messages, which can be nested (Section 4.3.1.1), microservice links, which define dependencies into configurations (Section 4.3.1.2), and message type dependencies, which allow tracking the termination of exchanges of messages when servicing a client call (Section 4.3.1.3).

4.3.1.1 Path of application messages

The application messages that are exchanged in the distributed application belong to three categories (requests, replies, and events) and implement two interaction modes (client-server and publish-subscribe). We are interested in modelling the reaction to application messages that leads to the execution of actions R_{ms} of Algorithm 1.

In the client-server interaction mode, a caller sends a request message to the callee and the caller waits for the reply from the callee. The service of the transaction request by the callee may lead to the sending of requests that form sub-transactions, that are also called nested transactions. In other words, there exist paths of messages.

In the publish-subscribe interaction mode, a producer asynchronously sends an event to consumers, whose identities are unknown to the producer, i.e. the routing is done by a broker according to a filtering mechanism. The consumption of an event may lead to the publication of new events, and according to the Event Collaboration pattern [Fowler, 2006], these events form a flow of events.

Of course, the two interaction modes can be mixed, such that the treatment of a transaction may lead to the production of events and the consumption of an event may lead to the sending of transaction requests.

It is important to observe that a client service call initiated by a message, being a request or an event, requires the execution of a termination detection algorithm. Following [Kramer and Magee, 1990], cycles in the path of messages are not forbidden, but we assume that the application is designed so that the execution of a path of messages is bounded and free of deadlocks.

Moreover, we assume that every message that starts a path of messages, being initiated by an actor or a microservice, is tagged with a collaboration identifier (similar either to a correlation identifier for transactions [Hohpe and Woolf, 2003], or to an asynchronous completion token for events [Schmidt et al., 2001], or else to the tagging of the so-called root transaction [Ma et al., 2011, Baresi et al., 2017]). When the message comes from an actor, it is the front-end F that tags the message with a new collaboration identifier. Thereafter, all the application messages of a collaboration are tagged with its collaboration identifier. We denote by $\mathit{cid}(m)$ the collaboration identifier of the application message m, by $\mathit{firstmsg}(\mathit{cid})$ the first message of collaboration cid, and by $\mathit{firstms}(\mathit{cid})$ the first microservice that receives $\mathit{firstmsg}(\mathit{cid})$.

4.3.1.2 Link dependencies

From the configuration in the instance model (presented in Section 3.2.2), we get the set S of microservices and the set L of links between microservices. These links constitute what we call the link dependencies. They are also called static dependencies in [Ma et al., 2011, Baresi et al., 2017], but we prefer avoiding using the terms “static” and “dynamic” dependencies because both are specified at design time in our approach. As in Chapter 3, two classes of links model the two types of connectors that we consider in our work.

Firstly, a client-server link connects a client microservice to a server microservice

through a chain of the following model elements: a client contract, which is attached to the client microservice, a client-server connector, and a server contract, which is attached to the server microservice. Thus, a client-server link is defined by a caller $ms_{caller} \in S$, a client contract $c_{cli} \in C_{cli}$, a server contract $c_{ser} \in C_{ser}$ and a callee $ms_{callee} \in S$, and we denote the set of client-server links by L_{cs}.

Secondly, a publish-subscribe link connects a producer microservice to a consumer microservice through a chain of the following model elements: a producer contract, which is attached to the producer microservice and contains a topic or routing key (resp. in MQTT and AMQP terminology) for the publication of events *via* this contract, a publish-subscribe connector, and a consumer contract, which is attached to the consumer microservice and contains a subscription filter or binding key (resp. in MQTT and AMQP terminology). Then, a publish-subscribe link is defined by a producer $ms_{prod} \in S$, a producer contract $c_{prod} \in C_{prod}$, a consumer contract $c_{cons} \in C_{cons}$ and a consumer $ms_{cons} \in S$. We denote the set of publish-subscribe links by L_{ps}. Finally, we have $L = L_{cs} \cup L_{ps}$.

Therefore, we model the link dependencies of a configuration in a directed multigraph $\mathcal{G}_L = (S, L, \phi_L)$, where S is the set of vertices, L is the set of edges, and ϕ_L is the incidence function mapping every edge to an ordered pair of vertices, here the function mapping the link $(ms_1, c_1, c_2, ms_2) \in L$ to the pair $(ms_1, ms_2) \in S^2$ with (c_1, c_2) being either in $C_{cli} \times C_{ser}$ or in $C_{prod} \times C_{cons}$. We denote by $\mathit{pathsTo}(ms)$ the set of paths to microservice ms using the set of edges L.

4.3.1.3 Message type dependencies

We complement the type model (presented in Section 3.2.1) with message type dependencies, i.e. we consider that application designers provide the set of message type dependencies that are typically extracted both from source code and sequence diagrams as follows.

Microservice types "receive" message types through server and consumer contract types; these types define the offered API of a microservice. For each message type of the offered API, architects and developers describe the **potential sequences** of message types that **may** be sent:

(i) They are sequences of message types that are exchanged *via* FIFO links so that offsets can be used to know the messages that may have already been received and the messages that may be received in the future;

(ii) There are several sequences of such message types so that the corresponding UML sequence diagram may contain `alt` fragments.

The description is then a conservative over-approximation of the set of messages to be sent when receiving a message of a given type. For instance, the expression $(\tau_1, \mu_1) ::= (\tau_2, \mu_2)(\tau_3, \mu_3)^2(\tau_4, \mu_4)|(\tau_2, \mu_2)(\tau_5, \mu_5)$ specifies that the microservice of type τ_1 receiving a message of type μ_1 may lead to either the sending of a message of type μ_2 to a microservice of type τ_2, then of two messages of type μ_3 to the same or two different microservices of type τ_3, and finally of a message of type μ_4 to a microservice of type τ_4, or the sending of a message of type μ_2 to a microservice of type τ_2 and of a message of type μ_5 to a microservice of type τ_5.

The number of consecutive repetitions is noted with an exponent ('$*$' for an unknown number of consecutive repetitions greater than 1) and is ignored if it is equal to 1. The reason for sequencing the sent message types and for adding the number of consecutive repetitions is to be capable of answering the following question: "may a microservice that has already been involved in a collaboration *cid* continue to participate in that collaboration (in the future) before *cid* ends?"

From the sequence diagram of Figure 4.1, we extract the following expressions describing what will happen after receiving the message type `attachFileToProject` by the microservice type `ProjectService` participant. The microservice type names are shortened to participant names with upper case letters, that is `ProjectService` into `PS`, etc., and we use `F` to name (the type of) the front-end. Observe that we must ignore local synchronous messages `isProjectLocked`, `findValidFileById`, and `addFileToProject`. Here follows the expressions:

```
(PS,attachFileToProject) ::= (AS,getLogin),
(AS,getLogin) ::= (PS, login),
(PS, login) ::= (PmS, checkUserPermission),
(PmS, checkUserPermission) ::= (US, findUser)^2,
(US, findUser) ::= (PmS, userFound)^2,
(PmS, UserFound) ::= (PS, permissionChecked) | (FS, permissionChecked),
(PS, permissionChecked) ::= (FS, doesFileExist),
(FS, doesFileExist) ::= (PmS, checkUserPermission),
(FS, permissionChecked) ::= (PS, fileExists),
(PS, fileExists) ::= (F, fileAttached).
```

These expressions give all the potential sequences of message types that may occur in the future when `ProjectService` receives the message `attachFileToProject`. Especially, the receipt of `checkUserPermission` by microservice type `PmS` may lead to the sending of two messages of type `findUser` to the same or two microservices of type `US` [1]. Similarly, the

1. It may be different microservices (instances) in the case of redundancy of the user service for tolerating failures (the state transfer or coordination between the replicas being out of the scope of our work).

receipt of `findUser` by microservice type `US` may lead to two messages of type `userFound` to the same or two microservices of type `PmS`. Observe also that `(PmS, userFound)` has two sequences, which we number `seq.1` and `seq.2` for `(PS, permissionChecked)` and `(FS, permissionChecked)`, respectively.

We denote by T the set of microservice types, by M the set of message types, and by a slight abuse of notation, F is also the type of the front-end. Therefore, a message type dependency is defined by the following terms:

1) a microservice type $\tau_r \in T$ that "receives" μ_r,
2) a "received" message type $\mu_r \in M$,
3) a "sent" message type $\mu_s \in M$ that τ_r "sends" in reaction to μ_r,
4) a microservice type or the front-end $\tau_s \in T \cup \{F\}$ that "receives" μ_s,
5) the alternative sequence (a number),
6) the offset in the alternative sequence, and
7) the number of consecutive repetitions *nrep* of the sending of μ_s.

We denote a message type dependency by $\delta = \Big((\tau_r, \mu_r), (\tau_s, \mu_s), seq, offset, nrep\Big)$ and the set of message type dependencies by Δ. For example, in the previous expression for `(PmS, UserFound)`, the offset of `(FS, permissionChecked)` in the second sequence ($seq = 2$) is 1 ($offset = 1$).

Hereafter, we build the directed multigraph $\mathcal{G}_\Delta = \Big((T \cup \{F\}) \times M, \Delta, \phi_\Delta\Big)$, where $(T \cup \{F\}) \times M$ is the set of vertices, Δ is the set of edges, and ϕ_Δ is the incidence function mapping every edge to an ordered pair of vertices, here the function mapping the message type dependency $\Big((\tau_r, \mu_r), (\tau_s, \mu_s), seq, offset, nrep\Big) \in \Delta$ to the pair $\Big((\tau_r, \mu_r), (\tau_s, \mu_s)\Big) \in T \times (T \cup \{F\})$.

At a given consistent distributed snapshot s, we assume that we can calculate the potential message types of the messages that a given microservice may receive or send to terminate the collaboration. More precisely, a solution can trace the collaborations in which a microservice participates and the messages that this microservice has already received and sent before s. In other words, from $\mathcal{G}_\Delta$ of configuration $\mathcal{C}$, when an application message m_r, with $m_r.type = \mu_r$, is sent by ms_r, with $ms_r.type = \tau_r$, to ms_s, with $ms_s.type = \tau_s$, by knowing $cid(m)$, it is possible to calculate the potential message types, denoted by $pmsgt(cid(m))$, that may be used for the treatment of m. For this calculation, the numbers *seq* and *nrep* are used (see below in the example).

Figure 4.4 depicts the multigraph of message type dependencies for the use case "attach a file to a project" of Figure 4.1. The triples on the edge of the

graph represent the three last terms $(seq, offset, nrep)$ of the message type dependency. From this multigraph, we can calculate the paths reachable from a given vertice. For example, when receiving message `userFound` after snapshot pms_2, we have `permissionChecked` $\in$ $pmsgt(cid($`userFound`$))$. As another example, at snapshots pms_3 or pms_4, the edge from (`PmS`,`checkUserPermission`) to (`US`,`findUser`)) has been crossed once, and by $nrep = 2$ we can deduce that $pmsgt(cid($`findUSer`$)) =$ {`userFound`, `permissionChecked`, `fileAttached`, `doesFileExist`, `checkUserPermission`, `findUser`}, i.e. microservice *pms* may be called in the future in the context of the same collaboration. Whereas, at snapshot pms_5, the edge from (`PmS`,`checkUserPerm.`) to (`US`,`findUser`) has been crossed once, and by $nrep = 2$ we can deduce that $pmsgt(cid($`findUSer`$)) =$ {`userFound`, `permissionChecked`, `fileAttached`, `doesFileExist`, `checkUserPerm.`}, i.e. *pms* will no more participate in the future to the collaboration.

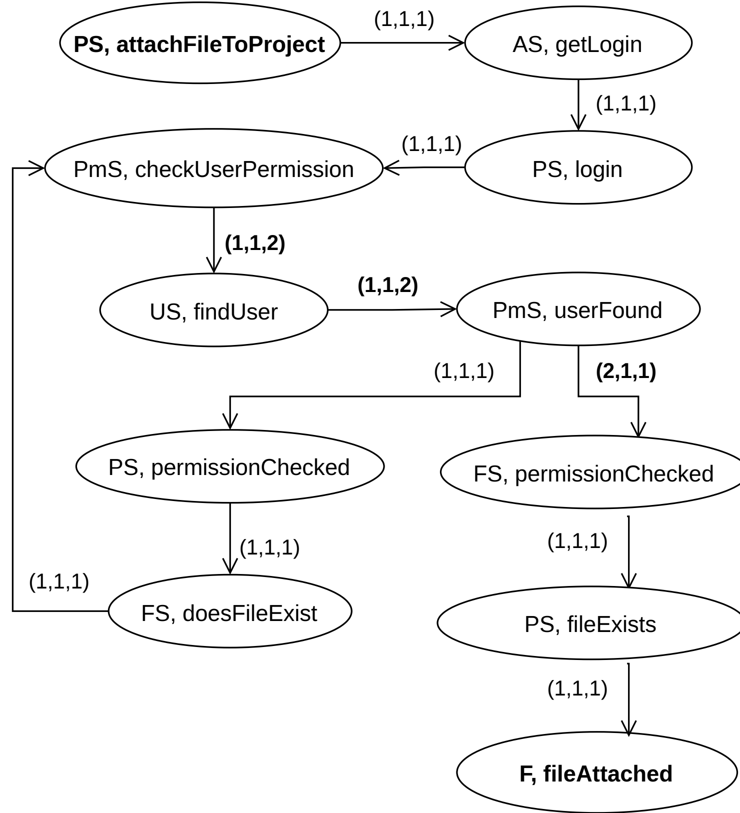


Figure 4.4 – Multigraph of message type dependencies when receiving the message type `attachFileToProject` in the use case `AttachFileToProject` as modelled in Figure 4.1

Consequently, since we presented in Section 4.3.1.1 the collaboration as an abstraction of a graph of messages (requests, replies, and events), we can define the specification of a configuration as follows.

Definition 1 (Specification of a configuration)**.** The specification $\mathbb{S}_{\mathcal{C}}$ of configuration $\mathcal{C}$ is composed of the directed multigraph of message type dependencies $\mathcal{C}.type.\mathcal{G}_{\Delta}$ ($\mathcal{C}.\mathcal{G}_{\Delta}$ in short) of its configuration type $\mathcal{C}.type$, the set of microservices $\mathcal{C}.S$, the set of links $\mathcal{C}.L$, and the directed multigraph of link dependencies $\mathcal{C}.type.\mathcal{G}_L$ ($\mathcal{C}.\mathcal{G}_L$ in short), i.e. $\mathbb{S}_{=}(\mathcal{C}.type, \mathcal{C}.\mathcal{G}_{\Delta}, \mathcal{C}.S, \mathcal{C}.L, \mathcal{C}.\mathcal{G}_L)$.

4.3.2 Adding continuity of service

Since a dynamic software update is expressed in terms of configurations, i.e. a source configuration $\mathcal{C}$ to a target configuration $\mathcal{C}'$, we do not limit ourselves to the version changes of only one microservice. Architects may want to specify service continuity: some replacement microservices and links are created before the old versions are removed so that there is always an instance, running either the old or new version, that is available and ready to receive messages. This information should also be of interest to system administrators in order to bring into play deployment strategies such as Blue Green deployment [Fowler, 2010] or Canary release [Sato, 2014]. But, deployment strategies are out of the scope of this thesis.

To express continuity of service requirements, software architects provide the following information in the software update request for a new configuration $\mathcal{C}'$:

1) The multigraphs $\mathcal{C}'.\mathcal{G}_L$ of the link dependencies and $\mathcal{C}'.\mathcal{G}_{\Delta}$ of the message type dependencies, the multigraphs $\mathcal{C}.\mathcal{G}_L$ and $\mathcal{C}.\mathcal{G}_{\Delta}$ being already known.
2) The set $\mathcal{R}_S$ of pairs of microservices (ms, ms') indicating that microservice ms in configuration $\mathcal{C}$ is being replaced by microservice ms' in configuration $\mathcal{C}'$.
3) Assuming that $ms \neq ms'$, the set $\mathcal{R}_L$ of pairs of links $\big((ms_o, c_o, c, ms), (ms_o, c'_o, c', ms')\big)$ indicating that link (ms_o, c_o, c, ms) in configuration $\mathcal{C}$ is being replaced by link (ms_o, c'_o, c', ms') in $\mathcal{C}'$.

In $\mathcal{R}_L$, the case $ms = ms'$, which corresponds to a connector change, is not considered here because we limit the study to microservice changes. Microservice ms that is linked with microservices ms_1, ms_2... that do not evolve is replaced by microservice ms'. The links between ms and ms_1, ms_2... are either replaced or removed. In case of replacement, these replacements belong to $\mathcal{R}_L$. We assume that the information provided in the software update request is correct by construction of the type and instance models at design time. The role of the replacement sets is to express service continuity as follows.

When a microservice is an entry point to the distributed application and changes version, the old version and the current version may run in parallel in an intermediate configuration so that, for instance, collaborations that are started in the source configuration can terminate, and collaborations are executed in the new configuration as

soon as the microservice of the new version is operational. Observe that there may exist intermediate and transient configurations between a source configuration and a target configuration.

Definition 2 (**Continuity of service**)**.** When transitioning from configuration $\mathcal{C}$ to configuration $\mathcal{C}'$ with replacement sets $\mathcal{R}_S$ and $\mathcal{R}_L$, a dynamic update satisfies service continuity if and only if the actors (through front-end F) or the neighbouring microservices always have access to one of the new or old versions of the microservices in $\mathcal{R}_S$ using links in $\mathcal{R}_L$.

When expressing continuity of service, we do not specify which version of a microservice takes part to a given collaboration, the old or new versions. For instance, can we change the version of a microservice while a collaboration is in progress? This depends on the definition of essential and non-essential changes that we now introduce from [Sokolowski et al., 2022].

4.3.3 Essential and non-essential changes

Without dynamic software updating, because a configuration $\mathcal{C}$ conforms to its configuration type $\mathcal{C}.type$, by using $\mathcal{C}.\mathcal{G}_L$ and $\mathcal{C}.\mathcal{G}_\Delta$, a collaboration can safely be executed in $\mathcal{C}$. But in case of dynamic software updating, the configuration $\mathcal{C}$, the configuration type $\mathcal{C}.type$, and the multigraphs of links $\mathcal{C}.\mathcal{G}_L$ and of message type dependencies $\mathcal{C}.\mathcal{G}_\Delta$ may change during the execution of a collaboration. Note that nothing for the moment forces a collaboration *cid* to be fully executed in current configuration $\mathcal{C}$ or in next configuration $\mathcal{C}'$ with $\mathcal{C}'.type \neq \mathcal{C}.type$ and $\mathcal{C}'.\mathcal{G}_\Delta \neq \mathcal{C}.\mathcal{G}_\Delta$. This is the role of the DSU algorithm to control the execution of collaborations when updating.

According to [Sokolowski et al., 2022] and extrapolating from workflow tasks to microservice messages, "a change is an essential change for a [collaboration *cid*] if the possible execution of any future message [in the next configuration] is not guaranteed to produce the same resulting state and side effects as executing [*cid* in current configuration]". Therefore, observe that an essential change is one that changes the message type dependencies of a collaboration, and the dynamic software updating algorithm must consider essential changes of collaborations that have already started and that are not terminated when the updating is started.

By a slight abuse of language, we speak of a version change of a microservice instead of the more complete formulation "microservice type change from τ to τ' with replacement of a microservice *ms* of type τ by a new instance *ms*$'$ that conforms to the new type τ'". Microservice *ms* possesses offered APIs that specify the message types it

may receive. If ms of type τ may receive a message of type μ from a microservice of type τ_i in configuration $\mathcal{C}$, then the change is essential if this is no more the case of ms' of type τ' in configuration $\mathcal{C}'$. Likewise, ms possesses required APIs that specify the message types it can send. If ms can send a message of type μ in configuration $\mathcal{C}$ to a microservice of type τ_i, then the change is essential if this is no more the case of ms' of type τ' in configuration $\mathcal{C}'$. In a dual manner, message types can be added into $\mathcal{C}'$ and change the behaviour (of the specification). A dynamic update from $\mathcal{C}$ to $\mathcal{C}'$ is defined as the tuple $(\mathcal{C}, \mathbb{S}_{\mathcal{C}}, \mathcal{C}', \mathbb{S}_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$.

Definition 3 (Essential change). In the context of dynamic update $(\mathcal{C}, \mathbb{S}_{\mathcal{C}}, \mathcal{C}', \mathbb{S}_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$, a version change of microservice ms in configuration $\mathcal{C}$ and with $ms.type = \tau$ into microservice ms' in configuration $\mathcal{C}'$ and with $ms.type = \tau'$ is an essential change w.r.t. collaboration cid if a potential message type μ_{cid} that ms may send or receive when involved in cid no longer exists in $\mathcal{C}'.\mathcal{G}_\Delta$ as it is in $\mathcal{C}.\mathcal{G}_\Delta$ or appears as new in $\mathcal{C}'.\mathcal{G}_\Delta$. Formally,

$$
\begin{aligned}
& essentialchange(ms, cid) \;= \\
& \exists (ms, ms') \in \mathcal{R}_S,\ \exists \tau_i \in T,\ \exists \mu_{cid} \in pmsgt(cid), \\
& \quad \wedge\ ms.type = \tau \wedge ms'.type = \tau' \wedge \tau \neq \tau' \wedge \tau_i \in T' \\
& \quad \wedge \vee \oplus \exists \Big((\tau_i, *), (\tau, \mu_{cid}), *, *, * \Big) \in \mathcal{C}.\mathcal{G}_\Delta \\
& \qquad\quad \oplus \exists \Big((\tau_i, *), (\tau', \mu_{cid}), *, *, * \Big) \in \mathcal{C}'.\mathcal{G}_\Delta \\
& \qquad \vee \oplus \exists \Big((\tau, \mu_{cid}), (\tau_i, *), *, *, * \Big) \in \mathcal{C}.\mathcal{G}_\Delta \\
& \qquad\quad \oplus \exists \Big((\tau', \mu_{cid}), (\tau_i, *), *, *, * \Big) \in \mathcal{C}'.\mathcal{G}_\Delta
\end{aligned}
$$

As a first example, let the microservice fs of type `File Service` change version to remove authorisation. Considering $\mathcal{C}.\mathcal{G}_\Delta$ and $\mathcal{C}'.\mathcal{G}_\Delta$, removing the authorisation is equivalent to removing the edge from (`FS`,`doesFileExist`) to (`PmS`,`checkUserPerm.`) plus removing the vertice (`FS`,`permissionChecked`). If fs is replaced at snapshot fs_2 then fs cannot consume message `permissionChecked` and the collaboration is in error.

As a second example, imagine that microservice ps of type `Project Service` is changed so that the authentication `getLogin` must be performed by a synchronous request. So, `(PS, attachFileToProject) ::= (AS, getLogin)` is changed into expression `(PS, attachFileToProject) ::= (AS, getLogin)(PmS, checkUserPermission)`. Considering $\mathcal{C}.\mathcal{G}_\Delta$ and $\mathcal{C}'.\mathcal{G}_\Delta$, this is equivalent to adding an edge from (`PS`,`attachFileToProject`) to (`AS`,`getLogin`) plus removing the vertice (`PS`,`Login`). If the replacement of ps occurs at snapshot ps_2, then ps cannot consume message `Login` and the collaboration is in error.

4.3.4 Correct dynamic update

First, recall that we assume that the same update, but performed *offline*, is correct and that, differently from [Ma et al., 2011, Baresi et al., 2017], a dynamic update transitions the distributed application from a source configuration to a target configuration, i.e. involving not only one microservice but a set of microservices. The definition must then consider microservices that are removed in addition to version-changing microservices.

In addition, since the approach is based on a model at runtime, depending on the strategy chosen for the transition, e.g. concurrent version [Baresi et al., 2017], the type model may contain both current and next microservice types. Accordingly, the DSU algorithm may transition the distributed application from a source configuration $\mathcal{C}$ to a target configuration $\mathcal{C}'$ via an intermediate configuration, denoted by $\mathcal{C}^+$, that instantiates both current and next types. Importantly, we assume that specifications $\mathbb{S}_{\mathcal{C}}$ and $\mathbb{S}_{\mathcal{C}'}$ are built so that $\mathbb{S}_{\mathcal{C}^+}$, which constrains $\mathcal{C}^+$, are correct, i.e. all the type models can be instantiated, and the corresponding application is functionally meaningful. We denote by $I(s)$ the set of collaborations in progress, a.k.a. ongoing collaborations, at a given snapshot. We consider that, using $\mathcal{C}.\mathcal{G}_\Delta$, every microservice is capable of knowing whether it has already participated in a collaboration $cid \in I(s)$, and if so, whether it may participate again to cid before the termination of the collaboration. More precisely, we can calculate the set $pmsgt(cid)$ of potential message types of the messages that a given microservice may receive or send to terminate the collaboration.

Next, using multigraph $\mathcal{C}.\mathcal{G}_\Delta$, the following information can be derived. We denote by $pp(\mathcal{C}.\mathcal{G}_\Delta, s, cid)$ the set of past participants (microservices) that have already participated in collaboration cid in the context of configuration $\mathcal{C}$ and before consistent snapshot s, and by $fp(\mathcal{C}.\mathcal{G}_\Delta, s, cid)$ the set of potential future participants (microservices) necessary to terminate collaboration cid in the context of configuration $\mathcal{C}$ and from consistent snapshot s (this set does not include microservices that have already participated in cid before s and that will no more participate). For example, by the expression `(US, findUser)`, microservice *pms* knows that it may participate twice by receiving a message `userFound`.

In order to formalise the various situations using consistent distributed snapshots, we label some of them. s_r denotes the logical time of the replacement of microservices, e.g. $s_r[ms]$ and $s_r[ms']$ for the replacement of ms by ms'. s_d denotes the logical time of the deletion of microservices, e.g. $s_d[ms]$ for the deletion of ms. In the case of creation, the new microservice is operational, but it is up to the architect or the system administrator to decide whether it receives application messages or not. In the case

of replacement, the state transfer from ms to ms' is performed, ms can be stopped, and ms' can receive application messages. We denote by $Cids$ the set of collaboration identifiers.

Definition 4 (Correct dynamic update). A dynamic update $d = (\mathcal{C}, \$_{\mathcal{C}}, \mathcal{C}', \$_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$ is correct if and only if:

— if ms is going to be removed then no ongoing collaborations may involve again ms from snapshot $s_d[ms]$,

— if ms is going to be replaced by ms' and the version change is essential for an ongoing collaboration, say cid, then no ongoing collaborations, say cid_i, are such that ms has already participated in cid_i at snapshot $s_r[ms]$ and may again be involved to cid_i from snapshot $s_r[ms]$.

Formally,

$$
\begin{aligned}
& correct(d) \;=\; \forall ms \in \mathcal{C}.S, \\
& \wedge\; ms \notin \mathcal{C}'.S \implies \nexists cid \in Cids, ms \in fp(\mathcal{C}.\mathcal{G}_\Delta, s_d, cid) \\
& \wedge\; \wedge\, \exists (ms, ms') \in \mathcal{R}_S \\
& \quad \wedge\, \exists cid \in I(s), essentialchange(ms, cid) \\
& \quad \implies \forall cid_i \in I(s),\; \wedge\; ms \notin pp(\mathcal{C}.\mathcal{G}_\Delta, s_r, cid_i) \\
& \qquad\qquad\qquad\qquad\quad \wedge\; ms \notin fp(\mathcal{C}.\mathcal{G}_\Delta, s_r, cid_i)
\end{aligned}
$$

In the use case “attach a file to a project” modelled in Figure 4.1, microservice pms cannot be deleted at snapshot $s = (ps_4, as_2, pms_4, us_2, fs_2)$ because $pms \in fp(\mathcal{C}.\mathcal{G}_\Delta, s, cid(\text{attachFileToProject}))$. Concerning microservice ps, it cannot be replaced at snapshot ps_2 if the change is essential, for example, as seen previously, in the case of the new version assuming that the `getLogin` asynchronous message is replaced by a client-server request.

4.3.5 Snapshot-based definitions of the update conditions

In this section, the three safe state-of-the-art update conditions that we are interested in, i.e. quiescence (Section 4.3.5.1), freeness (Section 4.3.5.2) and essential freeness (Section 4.3.5.3), are reformulated following our snapshot-based approach.

4.3.5.1 Quiescence

According to [Kramer and Magee, 1990] and extrapolating from transaction dependencies to message dependencies, DSU algorithm involves the establishment of a quiescence region that is specified as the set of microservices that must remain passive, not participate in ongoing collaborations, and not initiate collaborations during the update, i.e. these microservices are blocked. Given a microservice ms, this set is

composed of the microservices that belong to paths of message dependencies in $\mathcal{G}_L$ that include *ms*.

Definition 5 (Quiescence). In the context of dynamic update $(\mathcal{C}, \mathbb{S}_{\mathcal{C}}, \mathcal{C}', \mathbb{S}_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$ and at consistent distributed snapshot s, microservice ms of configuration $\mathcal{C}$ is quiescent if ms and all the microservices that can directly or transitively send messages to ms are passive, and if none of these microservices has initiated a collaboration that is not terminated. Formally, $quiescent(ms) = \forall ms_i \in pathsTo(ms) \cup \{ms\},\ passive(ms_i) \wedge \big(\nexists cid \in I(s), firstms(cid) = ms_i\big)$

For example, in Figure 4.1, microservice *pms* depends upon microservice *ps* (through message `checkUserPermission`), i.e. $ps \in pathsTo(pms)$, *ps* itself depends upon *as* (through `login`), etc. Indeed, we have $pathsTo(pms) = \{ps, as, pms, us, fs\}$, and then *pms* is quiescent before the beginning and after the end of the collaboration, respectively at consistent distributed snapshots $(ps_1, as_1, pms_1, us_1, fs_1)$ and $(ps_5, as_4, pms_6, us_4, fs_4)$.

4.3.5.2 Freeness

According to [Ma et al., 2011, Baresi et al., 2017] and again extrapolating from transaction dependencies to message dependencies, to be free at the global state locating the deletion or replacement of the *ms* microservice, *ms* must not be involved in any ongoing collaboration that might require its participation again. Freeness implies that a collaboration is entirely executed either in the context of source configuration $\mathcal{C}$ or in the context of target configuration $\mathcal{C}'$.

Definition 6 (Freeness). In the context of dynamic update $(\mathcal{C}, \mathbb{S}_{\mathcal{C}}, \mathcal{C}', \mathbb{S}_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$ and at consistent distributed snapshot s, microservice ms of configuration $\mathcal{C}$ is free if it is passive, and no ongoing collaboration has already involved ms at s and may involve again ms from s. Formally, $free(ms) = passive(ms) \wedge \nexists cid \in I(s), ms \in fp(\mathcal{C}.\mathcal{G}_\Delta, s, cid)$.

For example, in Figure 4.1, the participation of *pms* in the collaboration starts after snapshot ps_2 and the sending of `checkUserPermission`, and ends, in case of failure, after snapshot pms_3, and in case of success, after snapshot pms_5. Therefore, compared to quiescence, *pms* is also free before consistent distributed snapshot $(ps_3, as_2, pms_1, us_1, fs_1)$, and after consistent distributed snapshot $(ps_4, as_2, pms_5, us_3, fs_2)$.

4.3.5.3 Essential freeness

In [Sokolowski et al., 2022], the authors add the distinction between essential and non-essential changes. Updating microservice *ms* with non-essential changes can be done as soon as *ms* is passive, and updating *ms* with essential changes must be done in a version consistent manner.

Definition 7 (Essential-Freeness). In the context of dynamic update $(\mathcal{C}, \$_{\mathcal{C}}, \mathcal{C}', \$_{\mathcal{C}'}, \mathcal{R}_S, \mathcal{R}_L)$ and at consistent distributed snapshot s, microservice ms of configuration $\mathcal{C}$ is essentially free if it is passive, and it is free or its change is non-essential w.r.t. the ongoing collaborations at s. Formally, $\mathit{essentiallyFree}(ms) = \mathit{passive}(ms) \wedge \big(\mathit{free}(ms) \vee \forall cid \in I(s), \neg \mathit{essentialchange}(ms, cid)\big)$.

For example, in Figure 4.1, compared to freeness, if the change is non-essential, pms is also essentially free for instance at consistent distributed snapshots $(ps_3, as_2, pms_2, us_2, fs_1)$, and $(ps_4, as_3, pms_4, us_3, fs_2)$.

4.4 Snapshot-Based DSU Algorithm

The role of the DSU algorithm is, first, to compute the set of ongoing collaborations at a given consistent distributed snapshot and detect the termination of collaborations (Section 4.4.1), second, to maintain the update condition while managing the reconfiguration (Section 4.4.2). The central entity of our solution is the autonomic manager. The autonomic manager should block microservices as little and as briefly as possible. This is why, in addition to the three different update conditions, four updating strategies are presented. Our snapshot-based DSU algorithm is safe because we verify by construction that every dynamic update is a correct one (cf. Definition 4).

4.4.1 Termination detection of collaborations

A collaboration cid is terminated when all the microservices and the front-end that may be involved in cid are passive and no application messages about cid are in transit. Snapshot-based distributed detection algorithms count the number of application messages that are sent and received, and build a consistent distributed snapshot to get these counters in order to check whether the collaboration is terminated [Chandy and Lamport, 1985, Dijkstra, 1987]. It is possible to build snapshots only when the termination of the collaborations must be asserted or periodically. Consistent distributed snapshots are initiated by the autonomic manager and triggered in Action M of Algorithm 2. We do not include the algorithm of front-end F, as it is very similar.

For each global state of the distributed system, we denote the number of application messages that are in transit for collaboration cid by $\mathit{inTransit}[cid]$. The termination of cid is detected when all microservices plus F are passive and $\mathit{inTransit}[cid]$ equals 0. To count the number of in-transit messages, every microservice ms and F are equipped with counters of application messages mc, one counter per collaboration identifier, e.g. $mc_{ms}[cid]$ and mc_F. The microservices plus F collectively maintain the invariant P_1

defined as:
$P_1 : \forall cid \in Cids, inTransit[cid] = mc_F + \Sigma_{ms \in S}\, mc_{ms}[cid]$

Invariant P_1 is obtained when $mc[cid]$ are initially 0 and by the following rule (with a similar rule for F):

— Rule 1: When microservice *ms* sends (resp. receives) an application message for collaboration *cid*, it increments (resp. decrements) $mc_{ms}[cid]$.

The autonomic manager, microservices and F possess a snapshot counter, which the autonomic manager increments when launching a new consistent distributed snapshot. The autonomic manager broadcasts a SNAPSHOT message to all microservices plus F. The microservices obey the following rules (with similar rules for F):

— Rule 2: Every application message is tagged with the snapshot counter of the sending microservice.

— Rule 3: A microservice takes a snapshot and sends a SNAPSHOT message to the autonomic manager once per distributed snapshot and as soon as it receives either a SNAPSHOT message from the autonomic manager or an application message that is tagged with a snapshot counter of a greater value.

— Rule 4: A microservice only sends a SNAPSHOT message when it is passive.

The counting of the number of application messages is per collaboration, and the sum in Invariant P_1 is computed by the autonomic manager from the values of the counters recorded by the microservices and F. Clearly, the number of messages can be counted incrementally, i.e. from snapshot to snapshot as long as the following two conditions are respected:

— The autonomic manager does not start a distributed snapshot until the previous one is finished.

— The distributed snapshots are consistent: no application message is counted "received" and not "sent" in the global state that corresponds to the snapshot.

The pseudo-code at microservices for the application algorithm of DSU and the termination detection algorithm is written in Algorithm 2, which complements the basic Algorithm 1 (see Section 4.2.2).

Lemma 1. *Algorithm 2 builds a consistent distributed snapshot in finite time.*

Proof. (Sketch) Since the autonomic manager broadcasts a SNAPSHOT message to every microservice and to F, by Rule 2, and by the fault-free hypothesis of the distributed system model, all microservices plus F eventually receive a SNAPSHOT message or an application message with an incremental snapshot counter. Therefore, by Rule 3, all microservices and F eventually take a snapshot of the counters and send it to the autonomic manager. So, the autonomic manager eventually receives a snapshot from

Algorithm 2 Microservice with termination detection—code at ms

```
 1: Local variables:
 2:     status ∈ {active, passive}                        // passive allows to snapshot
 3:     sc ∈ ℕ init 0                                     // Snapshot counter
 4:     mc ∈ Cids → ℕ init ∅             // Message counter of appli. msgs, per cid
 5: R: {⟨cid, sc_s, m⟩ has arrived from s ∧ status = passive}
 6:     receive ⟨cid, sc_s, m⟩ from s
 7:     mc[cid] ← mc[cid] − 1  (* Rule 1 *)
 8:     if sc < sc_s then
 9:         send ⟨SNAPSHOT, sc_s, mc⟩ to am  (* Rules 3 and 4 *)
10:         sc ← sc_s
11:     status ← active; treatment of m, including calls to S
12: S: {⟨cid, sc, m⟩ is being sent to d ∧ status = active}
13:     mc[cid] ← mc[cid] + 1  (* Rule 1 *)
14:     send ⟨cid, sc, m⟩ to d  (* Rule 2 *)
15: I: {status = active}
16:     status ← passive
17: M: {⟨SNAPSHOT, sc_am⟩ has arrived from am ∧ status = passive}
18:     receive ⟨SNAPSHOT, sc_am⟩ from am
19:     if sc < sc_am then
20:         send ⟨SNAPSHOT, sc_am, mc⟩ to am  (* Rules 3 and 4 *)
21:         sc ← sc_am
```

all the microservices and F: the distributed snapshot is completed in finite time. In addition, by the FIFO order of the links, every application message sent after a local snapshot of the sender is received after the local snapshot of the receiver. Thus, the distributed snapshot is consistent. □

Theorem 2. *Algorithm 2 is a correct termination-detection algorithm for collaborations.*

Proof. (Sketch) By applying Rule 1, the algorithm is designed so that predicate P_1 is an invariant of the algorithm for every collaboration. By Lemma 1 and Rule 4, the evaluation of the predicate by the autonomic manager is done in a consistent way and using local snapshots at which all microservices and F are passive. Therefore, the autonomic manager eventually detects that there are no more messages in transit, and all microservices and F no more participate in the given collaboration, i.e. the termination of the collaboration is safely and eventually detected. □

4.4.2 DSU algorithm and updating strategies

In this section, we first outline the three main phases of our DSU algorithm (Section 4.4.2.1). Then, we present different versions of DSU algorithm for different update conditions with different reaching strategies: in Section 4.4.2.2 for quiescence and in Section 4.4.2.3 for freeness and essential freeness.

4.4.2.1 Overview of the DSU algorithm

We make sure that two executions of the DSU algorithm do not overlap one another. It is achieved simply by means of an execution counter that is incremented at each startup, and with control messages from previous executions being ignored. In the sequel of the section, DSU execution counters are omitted.

The DSU algorithm is decomposed into phases. Each phase is structured into a wave algorithm [Tel, 2002], i.e. each termination action of the wave at the autonomic manager is causally preceded by an action in each microservice. The initiator of the wave (phase) is the autonomic manager. The followers of the wave are the microservices plus front-end F. The properties of a wave algorithm are the following ones:

(1) Termination: Each wave is finite.

(2) Decision: Each wave contains at least one decide action.

(3) Dependence: In each wave, each decide action is causally preceded by an action in each follower.

The decide action of the wave is the end of the phase at the autonomic manager. The first receive action of the followers is the phase action of the microservice or front-end. The send action of a follower to the initiator is to declare the end of the contribution of the microservice or front-end to the phase. In addition, we also constrain the wave so that the first receive actions of the followers happen at a global state that is consistent, in order to build a consistent distributed snapshot, so that the detection of the property to reach is meaningful.

In the following, we present the different versions of the DSU algorithm according to the different properties to reach before updating (quiescence, freeness and essential freeness), and according to the different reaching strategies. When waiting for the update condition, the autonomic manager periodically launches a collaboration termination detection using Algorithm 2. Practically speaking, note that Algorithm 2 must be completed to trace the messages received in a collaboration by using the *offset* and *nrep* attributes of message type dependencies.

4.4.2.2 DSU algorithm for quiescence

The quiescence property is interesting when the message type dependencies are not or cannot be modelled, i.e. the DSU algorithm can only rely on $\mathcal{G}_L$ and $\mathcal{G}'_L$. In that situation, practically speaking, since the updating is expensive, we assume that the replacement set $\mathcal{R}_S$ specifies a set of microservices that must be replaced together, i.e. atomically w.r.t. collaborations. In other words, quiescence is used when there is no

other way to updating and by defining the smallest set of microservices to be jointly evolved. Clearly, if new collaborations that may involve microservices to be replaced are blocked by the front-end F, then the quiescence property becomes a stable property. Then, let define the quiescent set $qs(ms)$ as the set of microservices to passivate in order to reach the condition $quiescent(ms)$, thus we have $qs(ms) = \bigcup_{(ms,*)\in\mathcal{R}_S} quiescent(ms)$.

The DSU algorithm for quiescence is decomposed into the following phases:

P1) Inform all the microservices and the front-end F that a new instance of the DSU algorithm starts.

P2) Create the new microservices and the new microservice links for $\mathcal{C}'$.

P3) From now on, block new collaborations in order to passivate microservices in $\bigcup_{(ms,*)\in\mathcal{R}_S} quiescent(ms)$.

P4) When quiescence is reached for all the microservices to be replaced, perform all the replacements.

P5) Inform all the microservices and F to start execution in $\mathcal{C}'$.

P6) Delete microservices and links that have been replaced.

P7) Inform all the microservices and F that the updating is finished.

4.4.2.3 DSU algorithms for essential freeness and freeness

First of all, recall that essential freeness and freeness are local properties, i.e. they can be stated at the local snapshot of a microservice. However, they may not be stable properties, i.e. although every collaboration completes in finite time, there could always be collaborations running on a microservice.

Following [Baresi et al., 2017] and [Sokolowski et al., 2022], here is the list of strategies that we consider: "concurrent versions", "blocking messages" and "blocking collaborations" (*resp.* "blocking tasks" and "blocking instances" in the case of workflows [Sokolowski et al., 2022]), and "waiting (at most t seconds)".

Concurrent versions, blocking messages/collaborations updating strategies. In the "concurrent versions" strategy, microservice ms and its replacement ms' can run concurrently so that ms terminates the execution of ongoing collaborations and ms' takes care of new collaborations. Therefore, essential freeness or freeness become a stable property. It is worth noting that it is the only strategy that allows service continuity.

In the "blocking messages" strategy, microservice ms and its replacement ms' cannot run concurrently, e.g. they must operate exclusively. Locally, ms traces its participation in collaborations and is able to state whether it is free (*resp.* essentially free). For ms to stop participating in new collaborations, neighbouring microservices block messages

of new collaborations and for *ms*. Then, freeness or essential freeness becomes a stable property. The replacement may include a state transfer during which *ms* and ms' must synchronise.

In the “blocking collaborations” strategy, microservice *ms* and its replacement ms' cannot run concurrently. No new collaboration that may involve *ms* may be started so that freeness or essential freeness becomes a stable property. Again, the replacement may include a state transfer during which *ms* and ms' must synchronise.

The DSU algorithm for strategies “concurrent versions” and “blocking messages/-collaborations” is decomposed into the following phases:

P1) Inform all the microservices and the front-end F that a new execution of the DSU algorithm starts.

P2) Create the new microservices and the new microservice links for configuration $\mathcal{C}'$.

P3) From now on:
- — microservices that will disappear cannot initiate new collaborations, and
- — in strategy “concurrent versions”, new collaborations are executed in the context of configuration $\mathcal{C}'$ using the new microservices and microservice links,
- — in strategy “blocking messages”, microservices and F block messages of new collaborations and for microservices to replace,
- — in strategy “blocking collaborations”, microservices and F block new collaborations that may involve microservices to be replaced.

P4) When freeness (*resp.* essential freeness) is reached for microservices in $\mathcal{R}_S$ to be replaced, perform the replacements.

P5) Wait for all the replacements to be done in order to delete microservices and links that have been replaced.

P6) Inform all the microservices and the front-end F that the updating is finished.

Waiting (at most *t* seconds) updating strategy. A microservice *ms* and its replacement ms' cannot run concurrently. When the freeness (*resp.* essential freeness) property is true, *ms* tries to maintain the property up to the blocking of all incoming messages from its neighbours. If the synchronisation with neighbours succeeds, the microservice asks the autonomic manager for the replacement. As done in [Vandewoude et al., 2007], in order to ensure the liveness property of the updating in finite time, we switch after t seconds to a blocking solution, i.e. either the DSU algorithm for strategy “blocking messages” or “blocking collaborations”, or even the DSU algorithm for quiescence.

DSU algorithm for strategy “waiting” is composed of the following phases:

P1) Inform all the microservices and the front-end F that a new execution of the DSU algorithm starts.

P2) Create the new microservices and the new microservice links for configuration $\mathcal{C}'$.

P3) During at most t seconds, when freeness (*resp.* essential freeness) is reached at a microservice to be replaced, synchronise with neighbours to block all incoming messages, and if synchronisation succeeded (i.e. freeness or essential freeness still holds) perform the replacement.

P4) If some microservices have not been replaced before the deadline of t seconds, then move either to the DSU algorithm for blocking messages/collaborations at Phase 3 or to the DSU algorithm for quiescence at Phase 3.

P5) Delete microservices and links that have been replaced.

P6) Inform all the microservices and the front-end F that the updating is finished.

4.5 Discussion

In this chapter, we have tackled the problem of dynamic software updating at the configuration level for microservice-based applications. Using our complemented runtime model, the autonomic manager middleware service decides when the updating of microservices can be performed safely and how the impacted microservices are kept in a consistent state. In addition, such a middleware service can use different updating strategies to trade-off between characterising the impact of changes (e.g. whether essential or not) and minimising the duration of interruptions. The distributed application evolves incrementally from a configuration to another.

In [Ma et al., 2011, Baresi et al., 2017], the authors model what they call dynamic dependencies to trace the execution of transactions with past/future edges that are added or removed at runtime: past edges for already executed transactions and future edges for transactions that may be sent afterwards. In our work, we define message type dependencies from which we can calculate the potential past and future participants of collaborations. In addition, their work is limited to updating only one component at a time in transaction-based systems, where, in our work, a dynamic update concerns not only one microservice but a set of microservices, and not only transactions but also events.

In [Sokolowski et al., 2022], the developers manually tag component changes to be essential or not. In our proposal, developers complement the runtime model of Chapter 3, and we formalise the category of changes of being essential or not by computing microservice changes as the comparison of the set of received message types and of the set of sent message types (in reaction to the received message types). In addition, the workflow tasks can be called synchronously or asynchronously, then not limiting to client-server synchronous transactions. In our work, we bring into play

microservice architectures that contain both client-server (synchronous) interactions and publish-subscribe (asynchronous) interactions. We formulate essential freeness for these systems. In their work, the software architecture follows the Function-as-a-Service style for workflow-based applications. The DSU algorithm uses the pattern of diffusing computations of the workflows in order to add control messages sent by workflow engines that evaluate the update condition and can, for example, block components or messages.

In conclusion, previous DSU algorithms of the literature are dissemination-based: whether in periods without updating or during updating, they complement the application algorithm with control messages to detect the update condition [Ma et al., 2011, Baresi et al., 2017, Sokolowski et al., 2022]. These algorithms are optimal in the worst case.

For microservice-based architectures, we take a different approach: only periodically or when the update condition must be asserted, for instance because there is a reconfiguration request due to version changes, our DSU algorithm takes a periodic global snapshot of the distributed application. The rationale for this choice are the following: (1) microservice applications can be message-intensive so targeting the worst-case message complexity is limiting, (2) the bidirectional property of links cannot be obtained in a straightforward way with publish-subscribe connectors, and it is not reasonable to assume that brokers are configured for this purpose, (3) it is possible to express the update condition using consistent global system states, i.e. consistent distributed snapshots, and (4) the DSU algorithm, which uses primitives such as suspend/resume execution, create/delete microservices, link/unlink microservices, etc., is classically based on phases, which are naturally delimited by global system states, i.e. distributed snapshots.

Finally, there exist situations in which an entity is specifically designed to be composed with others, and both entities must evolve simultaneously. Apart from [Vandewoude et al., 2007], previous works [Ma et al., 2011, Baresi et al., 2017, Sokolowski et al., 2022] do not consider the version change of a set of microservices. In [Vandewoude et al., 2007], the problem is approached from the implementation point of view. In our work, we consider the situation at the time of specification, and we formalise the service continuity property. In addition, the knowledge of replacement sets to express service continuity should also be of interest to system administrators to enable deployment strategies such as blue/green deployment or canary release. Deployment strategies are out of the scope of this thesis, but will be in our future work.

4.6 Conclusion

To conclude this chapter, our contributions to the DSU problem for microservice-based applications are summarized as follows:

— We propose a general model for dynamic updating of microservice-based applications with client-server and publish-subscribe interactions. It includes link dependencies and message type dependencies that are built at design time and are part of the model at runtime.

— We give a definition of dynamic updates in terms of configurations, i.e. sets of microservices, not only one microservice. In order to express continuity of service, we also introduce two replacement sets of pairs of microservices instances and of communication links.

— We use a novel snapshot-based approach to define the correctness of a dynamic update in microservices, the concept of essential and non-essential changes, and the three state-of-the-art update conditions: namely quiescence, freeness, and essential freeness.

— We propose a snapshot-based DSU algorithm that is decomposed into phases and takes consistent distributed snapshots when necessary. The DSU algorithm is expressed for the three update conditions and for the four updating strategies of the literature: namely “concurrent versions”, “blocking messages”, “blocking collaborations”, and “waiting (at most t seconds)”. For instance, using “concurrent versions”, we ensure continuity of service.

In addition, a prototype of our DSU solution, and which is a proof of concept (not a middleware), is currently being implemented. It is named ARBORE (for “ARchitecting Based on microservice versiOns with REconfiguration”) and will be soon available at the following URL: `https://gitlabev.imtbs-tsp.eu/mimosae`.

A conclusion of our two contributions and some perspectives for future works are presented in the next chapter.

Chapter 5

Conclusion and Perspectives

We begin this concluding chapter with a summary of the main contributions of our work (Section 5.1). We then discuss some possible future works to extend our proposals (Section 5.2).

5.1 Contribution Summary

In this thesis, we have addressed the software evolution of microservice-based applications. We proposed solutions for modelling and tracking the evolutionary changes of microservice architectures, and for dynamically updating the microservice applications. The goal of our work and the industry requirements are summarised in two issues. On the one hand, we help engineers to more easily and more efficiently manage microservice version management by abstracting architectural evolution, tracing changes, and calculating reconfiguration plans. On the other hand, we tackle software updating at the level of the configuration of a running application, and we make the dynamic software updating approach more easily applicable and practical to use. Our main contributions have responded to the objectives as follows.

Specifying and modelling microservice-based application for version management. We have proposed runtime models for representing microservice-based applications and facilitating their version management. Our runtime model has two parts: one is the type model that describes the structural abstraction of microservice architectures, and the other is the instance model that captures the specific deployment configurations of microservice-based applications. Every instance conforms to a valid type. In the type model, we check that microservice types are instantiable. In the instance model, we check that microservice instances are deployable. These models are built at design time by engineers and used at runtime to mirror changes that occurr in the application,

following the “model at runtime” approach. The architectural elements to be versioned are provided in the type model with an identifier and a version number, which explicitly follows the syntactically well-defined SemVer policy. Then, deployable elements in the instance model that instantiate the types are also syntactically versioned. Our runtime models also support two communication mechanisms commonly-used in microservice architectures, namely client-server synchronous communication and publish-subscribe asynchronous communication.

Tracing evolutionary changes in microservice architectures. In order to trace evolutionary changes into microservice-based applications, we construct an evolution graph that records the trajectory and history of how the application evolves over time. Our evolution graph also has two parts: the first one is made of configuration type snapshots built from the type model, and the second one is made of configuration instance snapshots built from the instance model. As version changes are applied, new snapshots of the configuration type and of the configuration instance are created and committed to our evolution graph. Every configuration instance snapshot conforms to a configuration type snapshot. Practically speaking, an evolution is firstly traced in the type snapshots of the graph, i.e. new microservice implementations are added in the implementation repository, and, secondly, traced in the instance snapshots of the graph, i.e. a new configuration of the microservice application is prepared before being deployed. Thus, the evolution is considered as a transition from a source configuration to a target configuration. Engineered in a prototype called MIMOSAE, a semi-automatic MAPE-K control loop is implemented where our models act as its Knowledge base. An AI Planner is used to compute a plan from a given configuration to a target configuration, which contains a set of actions to be executed in order to reconfigure and redeploy the managed system. In this first prototype, client service calls are stopped during updating. Dynamic updating, i.e. while clients keep requesting the microservice application, is the subject of the second contribution.

Snapshot-based formulation of dynamic updating. In order to answer the questions of when dynamic updating can be performed safely and how the impacted microservices can reach and maintain a consistent state, we have first complemented our runtime models with a microservice execution model. This execution model includes microservice link dependencies that represent calling relationships between microservices in a configuration, and message type dependencies that are used to track the progress of message exchanges while servicing client calls. Then, microservice changes are characterised to be essential or non-essential by using the added execution model: roughly

speaking, an essential change is one that impacts message exchanges. Concerning continuity of service, we have introduced two replacement sets (pairs of microservices instances and pairs of communication links), and we have added a front-end entity between clients and microservices in order to smoothly switch from old version instances to new version instances. Then, with the introduction of the concept of consistent distributed snapshot, we have formalised the correctness of a dynamic update and the update conditions of the literature (quiescence, freeness, which allows version consistency, and essential freeness, which allows essential safety). As a consequence, it is possible to evaluate the update condition only when needed or periodically by taking photos (instead of systematically complementing all application exchanges between microservices with control messages). This explains why our second contribution is named after the concept of snapshot.

Wave-based DSU algorithm. The DSU algorithm not only ensures the correctness of dynamic updates by preserving the correct completion of ongoing and future client service calls (collaborations). The basis of the DSU algorithm is a distributed termination detection algorithm that is based on consistent distributed snapshots. Mainly, microservices count applications messages that are sent and received for each collaboration, and provide these counters as part of their snapshot. Then, the autonomic manager takes a consistent distributed snapshot and calculates termination detection by using the counters provided by the microservices. In the context of a consistent distributed snapshot, the autonomic manager evaluates the update conditions from the termination detection of collaborations. Next, the DSU algorithm is organised into waves: for instance, it informs all the microservices and the front-end that a new execution of the DSU algorithm starts, it creates the new microservices and the new links between microservices for the next configuration, it waits for the validation of the update condition before performing the reconfiguration, and finally it informs all the microservices and the front-end that the updating is finished. Importantly, when waiting for the update condition, the role of the DSU algorithm is to make the autonomic manager reconfigure the system so that the update condition becomes a stable property: once reached, the update condition still holds up to performing the reconfiguration (due to version changes). This is what is called the update strategy, and we integrate the four update strategies of the literature ("concurrent versions", "blocking messages", "blocking collaborations", and "waiting (at most t seconds)"). The DSU algorithm is currently being implemented in a prototype named ARBORE.

5.2 Future Work

We envisage the following future works.

Benchmarking microservices. We implement prototypes to validate the feasibility of our proposals and to demonstrate our solution. After the complete implementation of the second contribution, we plan a more systematic evaluation of our proposals by running benchmarks. The authors of [Aderaldo et al., 2017] have defined a set of benchmark requirements for microservice architectures and compared some open source candidates. Some existing works ([Gan et al., 2019, Grambow et al., 2020]) have proposed their approaches for defining microservice benchmarks. We can apply benchmarking to compare our contributions with other state-of-the-art solutions, for the metrics of message complexity, update timeliness (the delay to reach the update condition), and update duration (the overall time of an update).

Supporting more asynchronous communication patterns. Our work considers client-server interaction mode for synchronous communication and publish-subscribe pattern for asynchronous communication. In the field of distributed event-based systems, we could also consider complex event processing and streaming, which are not considered in this thesis. Therefore, in our future work, we plan to refine the models by adding other asynchronous communication modes (such as stream communication), and position technologies such as Kafka, which provides more than one communication mode, or such as service mesh platforms (like Istio), which bring into play software defined networking entities in microservice architectures.

Evolution beyond business microservices. In this thesis, we only consider the update of business microservices, but not the update of other parts of the system. For example, we could consider the version changes of the database systems or of the publish-subscribe connectors. The particularity of the changes is that infrastructure elements are usually based on external platforms or technologies from third-parties. Supporting such updates constitutes another perspectives of our work.

Deployment strategies. The knowledge of replacement sets to express service continuity should be of interest to system administrators to enable deployment strategies such as blue/green deployment or canary release. These deployment strategies are out of the scope of this thesis and can be introduced in future work. Some existing management tools are developed to automatically manage the deployment and execution of microservices that are wrapped into containers or other technology specific abstraction,

e.g. Kubernetes or Cloud Foundry. But these tools are not aware of the behaviours of the application. On this aspect, the work of [Tao, 2019, Boyer et al., 2018] provides a starting point to use a declarative architecture-based approach to deploy microservices on PaaS platforms by applying different deployment strategies.

Fault tolerance. Our contributions assume that the system is fault-free. Microservices, connectors, etc. are not subject to failures, nor are the front-end and the autonomic manager. We can start by considering simple faults such as crashes [Avizienis et al., 2004]. This last future work can be divided into two directions. On the one hand, we can consider microservice faults that appear after updating and try to mask them [Gärtner, 1999]. For example, a microservice may crash just after an update, thus indicating that the new microservice version is not bug-free. In this case, the new microservice is monitored to observe whether there is a failure just after updating. The autonomic manager can then launch a rollback process to the previous configuration, to an intermediate configuration, etc. The rollback policies can be leveraged by using the evolution graph: several branches, rollback to a different branch, etc. On the other hand, faults may happen during the execution of the DSU algorithm. In this case, we can roll back to a previous wave of the DSU algorithm, etc. [Elnozahy et al., 2002]

Bibliography

[OAS, 2007] (2007). Web Services Business Process Execution Language Version 2.0 - OASIS Standard. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html. Accessed: 2022-07-21.

[Mim, 2022] (2022). Mimosae project on gitlab. https://gitlabev.imtbs-tsp.eu/mimosae/mimosae.

[Aderaldo et al., 2017] Aderaldo, C. M., Mendonça, N. C., Pahl, C., and Jamshidi, P. (2017). Benchmark requirements for microservices architecture research. In *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pages 8–13. IEEE.

[Ahmed et al., 2020] Ahmed, B. H., Lee, S. P., Su, M. T., and Zakari, A. (2020). Dynamic software updating: a systematic mapping study. *IET Software*, 14(5):468–481.

[Akbulut and Perros, 2019] Akbulut, A. and Perros, H. G. (2019). Software versioning with microservices through the api gateway design pattern. In *2019 9th International Conference on Advanced Computer Information Technologies (ACIT)*, pages 289–292. IEEE.

[Alex, 2022] Alex, N. (2022). What Is a Rolling Release? https://www.easytechjunkie.com/what-is-a-rolling-release.htm. last accessed: 2022-09-05.

[Alonso et al., 2004] Alonso, G., Casati, F., Kuno, H., and Machiraju, V. (2004). Web services. In *Web services*, pages 123–149. Springer.

[AMQP Consortium, 2010] AMQP Consortium (2010). AMQP 1.0 revision 0, Recommendation. Standard, AMQP Consortium.

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE transactions on dependable and secure computing*, 1(1):11–33.

[Banijamali et al., 2020] Banijamali, A., Kuvaja, P., Oivo, M., and Jamshidi, P. (2020). Kuksa: Self-adaptive microservices in automotive systems. In *International Conference on Product-Focused Software Process Improvement*, pages 367–384. Springer.

[Barais et al., 2008] Barais, O., Meur, A. F. L., Duchien, L., and Lawall, J. (2008). Software architecture evolution. In *Software Evolution*, pages 233–262. Springer.

[Baresi et al., 2017] Baresi, L., Ghezzi, C., Ma, X., and La Manna, V. P. (2017). Efficient Dynamic Updates of Distributed Components Through Version Consistency. *IEEE Trans. on Software Eng.*, 43(4):340–358.

[Barnes et al., 2013] Barnes, J., Pandey, A., and Garlan, D. (2013). Automated planning for software architecture evolution. In *Proc. 28th IEEE/ACM ASE*.

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

[Bencomo et al., 2019] Bencomo, N., Götz, S., and Song, H. (2019). Models@ run. time: a guided tour of the state of the art and research challenges. *Software & Systems Modeling*, 18(5):3049–3082.

[Beugnard et al., 1999] Beugnard, A., Jezequel, J.-M., Plouzeau, N., and Watkins, D. (1999). Making components contract aware. *IEEE Computer*, 32(7).

[Blair et al., 2009] Blair, G., Bencomo, N., and France, R. B. (2009). Models@ run. time. *Computer*, 42(10):22–27.

[Bogner et al., 2019] Bogner, J., Fritzsch, J., Wagner, S., and Zimmermann, A. (2019). Assuring the evolvability of microservices: insights into industry practices and challenges. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 546–556. IEEE.

[Boyer et al., 2018] Boyer, F., de Palma, N., Tao, X., and Etchevers, X. (2018). A Declarative Approach for Updating Distributed Microservices. In *Proc. of the 40th Int. Conf. on Software Engineering: Companion Proceeedings*, ICSE ’18, pages 392–393, Gothenburg, Sweden.

[Buisson et al., 2016] Buisson, J., Dagnat, F., Leroux, E., and Martinez, S. (2016). Safe reconfiguration of coqcots and pycots components. *Journal of Systems and Software*, 122:430–444.

[Chandy and Lamport, 1985] Chandy, K. M. and Lamport, L. (1985). Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75.

[Chapman, 2014] Chapman, D. (2014). Introduction to DevOps on AWS. *Amazon Web Services*.

[Chappell, 2004] Chappell, D. (2004). *Enterprise service bus: Theory in practice*. O'Reilly.

[Chiba, 2000] Chiba, S. (2000). Load-time structural reflection in java. In *European Conference on Object-Oriented Programming*, pages 313–336. Springer.

[Computing et al., 2006] Computing, A. et al. (2006). An architectural blueprint for autonomic computing. *IBM White Paper*, 31(2006):1–6.

[Conway, 1968] Conway, M. (1968). How do committees invent. *Datamation*, 14(4):28–31.

[Cook et al., 2001] Cook, S., He, J., and Harrison, R. (2001). Dynamic and static views of software evolution. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001*, pages 592–601. IEEE.

[Cugola and Margara, 2012] Cugola, G. and Margara, A. (2012). Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Computing Surveys (CSUR)*, 44(3):15:1–15:62.

[de Giacomo et al., 2021] de Giacomo, G., Lenzerini, M., Leotta, F., and Mecella, M. (2021). From Component-based Architectures to Microservices: A 25-years-long Journey in Designing and Realizing Service-based Systems. In *Next-Gen Digital Services. A Retrospective and Roadmap for Service Computing of the Future*, pages 3–15. Springer.

[De Palma et al., 2001] De Palma, N., Laumay, P., and Bellissard, L. (2001). Ensuring dynamic reconfiguration consistency. In *Proc. 6th International ECOOP Workshop on Component-Oriented Programming*, pages 18–24.

[Decan and Mens, 2019] Decan, A. and Mens, T. (2019). What do package dependencies tell us about semantic versioning? *IEEE TOSE*.

[Dijkstra, 1987] Dijkstra, E. W. (1987). Shmuel Safra's version of termination detection. EWD—Note 998. circulated privately, http://www.cs.utexas.edu/users/EWD/ewd09xx/EWD998.PDF.

[Dijkstra and Scholten, 1980] Dijkstra, E. W. and Scholten, C. S. (1980). Termination detection for diffusing computations. *Information Proc. Letters*, 11(1):1–4.

[Dowling and Cahill, 2001] Dowling, J. and Cahill, V. (2001). Dynamic software evolution and the k-component model.

[Dragoni et al., 2017] Dragoni, N., Giallorenzo, S., Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.

[Ebert et al., 2016] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). DevOps. *IEEE Software*, 33(3):94–100.

[Elnozahy et al., 2002] Elnozahy, E., Alivisi, L., Wang, Y.-M., and Johnson, D. (2002). A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34(3):375–408.

[Erl, 2008] Erl, T. (2008). *SOA Design Patterns (paperback)*. Pearson Education.

[Erl, 2016] Erl, T. (2016). *Service-oriented architecture: Analysis and Design for Services and Microservices*. Pearson Education Incorporated Upper Saddle River.

[Esparrachiari et al., 2018] Esparrachiari, S., Reilly, T., and Rentz, A. (2018). Tracking and controlling microservice dependencies: Dependency management is a crucial part of system and software design. *Queue*, 16(4):44–65.

[Etzion and Niblett, 2011] Etzion, O. and Niblett, P. (2011). *Event Processing in Action*. Manning.

[Eugster et al., 2003] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A.-M. (2003). The Many Faces of Publish/Subscribe. *ACM Comp. Surveys*, 35(2).

[Feiler and Li, 1998] Feiler, P. and Li, J. (1998). Consistency in dynamic reconfiguration. In *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No. 98EX159)*, pages 189–196. IEEE.

[Florio and Di Nitto, 2016] Florio, L. and Di Nitto, E. (2016). Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 357–362. IEEE.

[Fowler, 2006] Fowler, M. (2006). Event Collaboration. https://martinfowler.com/eaaDev/EventCollaboration.html. last accessed: 2022-08-09.

[Fowler, 2010] Fowler, M. (2010). Blue Green Deployment. https://martinfowler.com/bliki/BlueGreenDeployment.html. last accessed: 2022-08-09.

[Fowler, 2018] Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[Francez, 1980] Francez, N. (1980). Distributed Termination. *ACM Transactions on Programming Languages and Systems*, 2(1):42–55.

[Gan et al., 2019] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., et al. (2019). An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18.

[Garlan, 2000] Garlan, D. (2000). Software architecture: a roadmap. In *Proc. of the International Conference on the Future of Software Engineering*, pages 91–101.

[Garlan et al., 2009] Garlan, D., Barnes, J. M., Schmerl, B., and Celiku, O. (2009). Evolution styles: Foundations and tool support for software architecture evolution. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 131–140. IEEE.

[Gärtner, 1999] Gärtner, F. C. (1999). Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26.

[Ghafari et al., 2012] Ghafari, M., Jamshidi, P., Shahbazi, S., and Haghighi, H. (2012). An architectural approach to ensure globally consistent dynamic reconfiguration of component-based systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 177–182.

[Godfrey and German, 2008] Godfrey, M. and German, D. (2008). The past, present, and future of software evolution. In *Proc. of the Conference on Frontiers of Software Maintenance*, pages 129–138. IEEE.

[Godfrey and German, 2014] Godfrey, M. and German, D. (2014). On the evolution of Lehman's Laws. *Journal of Software: Evolution and Process*, 26(7):613–619.

[Grambow et al., 2020] Grambow, M., Meusel, L., Wittern, E., and Bermbach, D. (2020). Benchmarking microservice performance: a pattern-based approach. In *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, pages 232–241.

[Grunske, 2005] Grunske, L. (2005). Formalizing architectural refactorings as graph transformation systems. In *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*, pages 324–329. IEEE.

[Haslum et al., 2019] Haslum, P., Lipovetzky, N., Magazzeni, D., and Muise, C. (2019). An introduction to the planning domain definition language. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 13(2):1–187.

[He et al., 2020] He, X., Tu, Z., Liu, L., Xu, X., and Wang, Z. (2020). Optimal evolution planning and execution for multi-version coexisting microservice systems. In *International Conference on Service-Oriented Computing*, pages 3–18. Springer.

[Hohpe and Woolf, 2003] Hohpe, G. and Woolf, B. (2003). *Enterprise Integration Patterns*. Addison-Wesley.

[Huang et al., 2009] Huang, G., Song, H., and Mei, H. (2009). Sm@rt: towards architecture-based runtime management of internetware systems. In *Proceedings of the First Asia-Pacific Symposium on Internetware*, pages 1–10.

[Huebscher and McCann, 2008] Huebscher, M. C. and McCann, J. A. (2008). A survey of autonomic computing – degrees, models, and applications. *ACM CSUR*, 40(3).

[Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.

[Huynh, 2017] Huynh, N. T. (2017). *A development process for building adaptative software architectures*. PhD thesis, Ecole nationale supérieure Mines-Télécom Atlantique.

[Jamshidi et al., 2018] Jamshidi, P., Pahl, C., Mendonça, N., Lewis, J., and Tilkov, S. (2018). Microservices: The journey so far and challenges ahead. *IEEE Software*, 35(3):24–35.

[Kawrykow and Robillard, 2011] Kawrykow, D. and Robillard, M. P. (2011). Non-essential changes in version histories. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 351–360. IEEE.

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50.

[Ketfi et al., 2002] Ketfi, A., Belkhatir, N., and Cunin, P.-Y. (2002). Automatic adaptation of component-based software. In *Second Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1365–1371. Citeseer.

[Killalea, 2016] Killalea, T. (2016). The Hidden Dividends of Microservices: Microservices aren't for every company, and the journey isn't easy. *ACM Queue*, 14(3):25–34.

[Kniesel et al., 2001] Kniesel, G., Costanza, P., and Austermann, M. (2001). Jmangler-a framework for load-time transformation of java class files. In *Proceedings First*

IEEE International Workshop on Source Code Analysis and Manipulation, pages 98–108. IEEE.

[Kramer and Magee, 1990] Kramer, J. and Magee, J. (1990). The evolving philosophers problem: dynamic change management. *IEEE TOSE*, 16(11).

[Kruchten, 1995] Kruchten, P. (1995). The 4 + 1 view model of architecture. *IEEE software*, 12(6):42–50.

[Lamport, 1978] Lamport, L. (1978). Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7).

[Lehman, 1980] Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076.

[Lehman and Ramil, 2002] Lehman, M. and Ramil, J. (2002). Software evolution and software evolution processes. *Annals of Software Engineering*, 14(1):275–309.

[Lehman et al., 1997] Lehman, M. M., Ramil, J., Wernick, P., Perry, D., and Turski, W. (1997). Metrics and laws of software evolution-the nineties view. In *Proc. of the 4th IEEE International Software Metrics Symposium*, pages 20–32. IEEE.

[Lewis and Fowler, 2014] Lewis, J. and Fowler, M. (2014). Microservices - a definition of this new architectural term. https://martinfowler.com/articles/microservices.html.

[Lim et al., 2015] Lim, L., Marie, P., Conan, D., Chabridon, S., Desprats, T., and Manzoor, A. (2015). Enhancing Context Data Distribution for the Internet of Things using QoC-awareness and Attribute-Based Access Control. *Annals of Telecommunications*, 71(3):121–132.

[Liu et al., 2020] Liu, L., He, X., Tu, Z., and Wang, Z. (2020). Mv4ms: A spring cloud based framework for the co-deployment of multi-version microservices. In *2020 IEEE International Conference on Services Computing (SCC)*, pages 194–201. IEEE.

[Ma et al., 2019] Ma, S.-P., Liu, I.-H., Chen, C.-Y., Lin, J.-T., and Hsueh, N.-L. (2019). Version-based microservice analysis, monitoring, and visualization. In *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pages 165–172. IEEE.

[Ma et al., 2011] Ma, X., Baresi, L., Ghezzi, C., Panzica La Manna, V., and Lu, J. (2011). Version-Consistent Dynamic Reconfiguration of Component-Based Distributed Systems. In *19th ACM SIGSOFT Symp. and 13th European Conf. on Foundations of Software Engineering*, pages 245–255, Szeged, Hungary.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. *ACM Sigplan Notices*, 22(12):147–155.

[Medvidovic and Taylor, 2010] Medvidovic, N. and Taylor, R. (2010). Software architecture: foundations, theory, and practice. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering*, volume 2, pages 471–472. IEEE.

[Mehta et al., 2000] Mehta, N., Medvidovic, N., and Phadke, S. (2000). Towards a Taxonomy of Software Connectors. In *22nd ACM ICSE*, Ireland.

[Méhus, J.-E. and Batista, T. and Buisson, J., 2012] Méhus, J.-E. and Batista, T. and Buisson, J. (2012). ACME vs PDDL: Support for Dynamic Reconfiguration of Software Architectures. In *French Conf. on Software Architectures (CAL)*, pages 48–57.

[Mens et al., 2003] Mens, T., Buckley, J., Zenger, M., and Rashid, A. (2003). Towards a taxonomy of software evolution. In *Proc. of the International Workshop on Unanticipated Software Evolution*, number CONF.

[Microsoft, 2022] Microsoft (2022). .Net: Free Cross-platform Open source – A developer platform for building all your apps. https://dotnet.microsoft.com/. Accessed: 2022-07-20.

[Miedes and Munoz-Escoı, 2012] Miedes, E. and Munoz-Escoı, F. D. (2012). Dynamic software update. *Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València, Technical Report ITI-SIDI-2012/004*.

[Moazami-Goudarzi, 1999] Moazami-Goudarzi, K. (1999). *Consistency preserving dynamic reconfiguration of distributed systems*. PhD thesis, Imperial College London (University of London).

[Morin et al., 2009] Morin, B., Barais, O., Jézéquel, J.-M., Fleurey, F., and Solberg, A. (2009). Models@ run. time to support dynamic adaptation. *Computer*, 42(10):44–51.

[Mühl et al., 2006] Mühl, G., Fiege, L., and Pietzuch, P. (2006). *Distributed Event-Based Systems*. Springer.

[Naur and Randell, 1969] Naur, P. and Randell, B. (1969). Software engineering: Report of a conference sponsored by the nato science committee, garmisch, germany, 7th-11th october 1968.

[Newman, 2015] Newman, S., editor (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

[OASIS, 2019] OASIS (2019). MQTT Version 5.0. Standard, OASIS Consortium.

[OMG, 2022] OMG (2022). Common Object Request Broker Architecture (CORBA). https://www.corba.org/. Accessed: 2022-07-20.

[Oracle, 2022] Oracle (2022). Java EE at a Glance. https://www.oracle.com/java/technologies/java-ee-glance.html. Accessed: 2022-07-20.

[Oreizy et al., 1998] Oreizy, P., Medvidovic, N., and Taylor, R. N. (1998). Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE.

[Parnas, 1971] Parnas, D. (1971). Information distribution aspects of design methodology.

[Peltz, 2003] Peltz, C. (2003). Web services orchestration and choreography. *IEEE Computer*, 36(10):46–52.

[Perry and Wolf, 1992] Perry, D. and Wolf, A. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52.

[Phung-Khac, 2010] Phung-Khac, A. (2010). *A model-driven feature-based approach to runtime adaptation of distributed software architectures*. PhD thesis.

[Preston-Werner, 2013] Preston-Werner, T. (2013). Semantic versioning 2.0.0. *http://semver.org*.

[Rajagopalan and Jamjoom, 2015] Rajagopalan, S. and Jamjoom, H. (2015). App–Bisect: Autonomous Healing for Microservice-Based Apps. In *Proc. of the 7th USENIX Workshop on Hot Topics in Cloud Computing*.

[Randell and Buxton, 1970] Randell, B. and Buxton, J. (1970). Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27th-31st October 1969.

[Ravichandran and Rothenberger, 2003] Ravichandran, T. and Rothenberger, M. (2003). Software reuse strategies and component markets. *Communications of the ACM*, 46(8):109–114.

[Raymond, 2003] Raymond, E. (2003). *The art of Unix programming*. Addison-Wesley.

[Saltzer et al., 1984] Saltzer, J. H., Reed, D. P., and Clark, D. D. (1984). End-to-end arguments in system design. *ACM Trans. on Comp. Syst.*, 2(4):277–288.

[Sampaio et al., 2017] Sampaio, A. R., Kadiyala, H., Hu, B., Steinbacher, J., Erwin, T., Rosa, N., Beschastnikh, I., and Rubin, J. (2017). Supporting Microservice Evolution. In *Proc. of the 33rd IEEE Int. Conf. Software Maintenance and Evolution*, pages 539–543.

[Sampaio et al., 2019] Sampaio, A. R., Rubin, J., Beschastnikh, I., and Rosa, N. S. (2019). Improving microservice-based applications with runtime placement adaptation. *Journal of Internet Services and Applications*, 10.

[Sato, 2014] Sato, D. (2014). Canary update strategies. https://martinfowler.com/bliki/CanaryRelease.html. last accessed: 2022-08-09.

[Schilling, 2000] Schilling, M. (2000). Toward a general modular systems theory and its application to interfirm product modularity. *Academy of management review*, 25(2):312–334.

[Schmidt et al., 2001] Schmidt, D.C., S., M., Rohnert, H., and Buschmann, F. (2001). *Pattern-Oriented Software Architecture: Volume 2, Patterns for Concurrent and Networked Objects*. Wiley.

[Seifzadeh et al., 2013] Seifzadeh, H., Abolhassani, H., and Moshkenani, M. S. (2013). A survey of dynamic software updating. *Journal of Software: Evolution and Process*, 25(5):535–568.

[Sindhgatta et al., 2010] Sindhgatta, R., Narendra, N., and Sengupta, B. (2010). Software evolution in agile development: a case study. In *Proc. of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 105–114.

[Sokolowski et al., 2022] Sokolowski, D., Weisenburger, P., and Salvaneschi, G. (2022). Change Is the Only Constant: Dynamic Updates for Workflows. In *44nd ACM Int. Conf. on Software Engineering*, Pittsburgh, PA, USA.

[Soni et al., 1995] Soni, D., Nord, R., and Hofmeister, C. (1995). Software architecture in industrial applications. In *Proc. of the 17th International Conference on Software Engineering*, pages 196–196. IEEE.

[Sorgalla et al., 2018] Sorgalla, J., Wizenty, P., Rademacher, F., Sachweh, S., and Zündorf, A. (2018). Ajil: enabling model-driven microservice development. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–4.

[Szyperski et al., 2002] Szyperski, C., Gruntz, D., and Murer, S. (2002). *Component software: beyond object-oriented programming*. Pearson Education.

[Tao, 2019] Tao, X. (2019). *Reliability of changes in cloud environment at PaaS level*. PhD thesis, Université Grenoble Alpes.

[Tel, 2002] Tel, G. (2002). *Introduction to Distributed Algorithms, 2nd Edition*, chapter Termination detection. Cambridge University Press.

[Thompson et al., 1997] Thompson, D., Exton, C., Garrett, L., Sajeev, A., and Watkins, D. (1997). Distributed component object model (DCOM). *Monash University, Department of Software Development, Melbourne, Australien.*

[Thönes, 2015] Thönes, J. (2015). Microservices. *IEEE software*, 32(1):116–116.

[Vandewoude et al., 2007] Vandewoude, Y., Ebraert, P., Berbers, Y., and D'Hondt, T. (2007). Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Trans. on Software Eng.*, 33(12):856–868.

[Vogel and Giese, 2010] Vogel, T. and Giese, H. (2010). Adaptation and abstract runtime models. In *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 39–48.

[Wang, 2019] Wang, Y. (2019). Towards service discovery and autonomic version management in self-healing microservices architecture. In *Proceedings of the 13th European Conference on Software Architecture-Volume 2*, pages 63–66, Paris, France.

[Wang et al., 2021] Wang, Y., Conan, D., Chabridon, S., Bojnourdi, K., and Ma, J. (2021). Runtime models and evolution graphs for the version management of microservice architectures. In *Proceedings of the 28th IEEE Asia-Pacific Software Eng. Conf.*, pages 536–541, Taipei, Taiwan.

[Waseem et al., 2020] Waseem, M., Liang, P., and Shahin, M. (2020). A systematic mapping study on microservices architecture in devops. *Journal of Systems and Software*, 170:110798.

[Wermelinger and Fiadeiro, 2002] Wermelinger, M. and Fiadeiro, J. L. (2002). A graph transformation approach to software architecture reconfiguration. *Science of Computer Programming*, 44(2):133–155.

[Yu and Mishra, 2013] Yu, L. and Mishra, A. (2013). An empirical study of Lehman's law on software quality evolution.

Appendix A

Implementation Details of Planner and Executor in MIMOSAE

A.1 Full PDDL file for MIMOSAE

Listing A.1 PDDL domain file for MIMOSAE

```
(define (domain microservice_planner_domain)
(:requirements :adl :typing)
(:types
    architecturalentity - object
    microservice - architecturalentity
    databasesystem - architecturalentity
    connector - architecturalentity
    pubsubconnector - connector
)
(:predicates
    (microservice ?m - microservice)
    (databasesystem ?db - databasesystem)
    (pubsubconnector ?c - pubsubconnector)
    (client_server_link ?mc - microservice ?ms - microservice)
    (producer_to_connector_link ?m - microservice ?c - pubsubconnector)
    (consumer_to_connector_link ?m - microservice ?p - pubsubconnector)
    (database_connection ?mc - microservice ?db - databasesystem)
)
(:action create_microservice
    :parameters (?m - microservice)
    :precondition (and (not (microservice ?m)))
    :effect (and (microservice ?m))
)
(:action create_databasesystem
    :parameters (?db - databasesystem)
    :precondition (and (not (databasesystem ?db)))
    :effect (and (databasesystem ?db))
)
(:action remove_microservice
    :parameters (?m - microservice)
    :precondition (and (microservice ?m))
    :effect (and (not (microservice ?m)))
)
```

```
34 (:action remove_databasesystem
35     :parameters (?db - databasesystem)
36     :precondition (and (databasesystem ?db))
37     :effect (and (not (databasesystem ?db)))
38 )
39 (:action link_client_server
40     :parameters (?mc - microservice ?ms - microservice)
41     :precondition (and (microservice ?mc)
42                        (microservice ?ms)
43                        (not (client_server_link ?mc ?ms)))
44     :effect (and (client_server_link ?mc ?ms))
45 )
46 (:action unlink_client_server
47     :parameters (?mc - microservice ?ms - microservice)
48     :precondition (and (microservice ?mc)
49                        (microservice ?ms)
50                        (client_server_link ?mc ?ms))
51     :effect (and (not (client_server_link ?mc ?ms)))
52 )
53 (:action create_pubsubconnector
54     :parameters (?c - pubsubconnector)
55     :precondition (and (not (pubsubconnector ?c)))
56     :effect (and (pubsubconnector ?c))
57 )
58 (:action remove_pubsubconnector
59     :parameters (?c - pubsubconnector)
60     :precondition (and (pubsubconnector ?c))
61     :effect (and (not (pubsubconnector ?c)))
62 )
63 (:action link_producer_to_pubsubconnector
64     :parameters (?m - microservice ?c - pubsubconnector)
65     :precondition (and (microservice ?m)
66                        (pubsubconnector ?c)
67                        (not (producer_to_connector_link ?m ?c)))
68     :effect (and (producer_to_connector_link ?m ?c))
69 )
70 (:action unlink_producer_from_pubsubconnector
71     :parameters (?m - microservice ?c - pubsubconnector)
72     :precondition (and (microservice ?m)
73                        (pubsubconnector ?c)
74                        (producer_to_connector_link ?m ?c))
75     :effect (and (not (producer_to_connector_link ?m ?c)))
76 )
77 (:action link_consumer_to_pubsubconnector
78     :parameters (?m - microservice ?c - pubsubconnector)
79     :precondition (and (microservice ?m)
80                        (pubsubconnector ?c)
81                        (not (consumer_to_connector_link ?m ?c)))
82     :effect (and (consumer_to_connector_link ?m ?c))
83 )
84 (:action unlink_consumer_from_pubsubconnector
85     :parameters (?m - microservice ?c - pubsubconnector)
86     :precondition (and (microservice ?m)
87                        (pubsubconnector ?c)
88                        (consumer_to_connector_link ?m ?c))
89     :effect (and (not (consumer_to_connector_link ?m ?c)))
90 )
91 (:action connect_microservice_to_databasesystem
92     :parameters (?ms - microservice ?dbs - databasesystem)
93     :precondition (and (microservice ?ms)
94                        (databasesystem ?dbs)
95                        (not (database_connection ?ms ?dbs)))
96     :effect (and (database_connection ?ms ?dbs))
97 )
```

```
98  (:action disconnect_microservice_from_databasesystem
99      :parameters (?ms - microservice ?dbs - databasesystem)
100     :precondition (and (microservice ?ms)
101                        (databasesystem ?dbs)
102                        (database_connection ?ms ?dbs))
103     :effect (and (not (database_connection ?ms ?dbs)))
104 )
```

Listing A.2 PDDL problem file for creating GDE architecture with synchronous client-server calls

```
1   (:domain microservice_planner_domain)
2     (:objects
3       DBUSi_DBUS_1_0_0 - databasesystem
4       DBPSi_DBPS_1_0_0 - databasesystem
5       DBFSi_DBFS_1_0_0 - databasesystem
6       DBPERMi_DBPERM_1_0_0 - databasesystem
7       PermSi_PermS_1_0_0 - microservice
8       Ai_Auth_1_0_0 - microservice
9       FSi_FS_1_0_0 - microservice
10      USi_US_1_0_0 - microservice
11      PSi_PS_1_0_0 - microservice
12      microservice_dummy - microservice
13    )
14    (:init
15      (microservice microservice_dummy)
16    )
17    (:goal
18      (and
19        (databasesystem DBUSi_DBUS_1_0_0)
20        (databasesystem DBPSi_DBPS_1_0_0)
21        (databasesystem DBFSi_DBFS_1_0_0)
22        (databasesystem DBPERMi_DBPERM_1_0_0)
23        (microservice PermSi_PermS_1_0_0)
24        (client_server_link PermSi_PermS_1_0_0 Ai_Auth_1_0_0)
25        (client_server_link PermSi_PermS_1_0_0 USi_US_1_0_0)
26        (database_connection PermSi_PermS_1_0_0 DBPERMi_DBPERM_1_0_0)
27        (microservice Ai_Auth_1_0_0)
28        (client_server_link Ai_Auth_1_0_0 USi_US_1_0_0)
29        (microservice FSi_FS_1_0_0)
30        (client_server_link FSi_FS_1_0_0 Ai_Auth_1_0_0)
31        (client_server_link FSi_FS_1_0_0 USi_US_1_0_0)
32        (client_server_link FSi_FS_1_0_0 PermSi_PermS_1_0_0)
33        (database_connection FSi_FS_1_0_0 DBFSi_DBFS_1_0_0)
34        (microservice USi_US_1_0_0)
35        (client_server_link USi_US_1_0_0 PermSi_PermS_1_0_0)
36        (client_server_link USi_US_1_0_0 Ai_Auth_1_0_0)
37        (database_connection USi_US_1_0_0 DBUSi_DBUS_1_0_0)
38        (microservice PSi_PS_1_0_0)
39        (client_server_link PSi_PS_1_0_0 FSi_FS_1_0_0)
40        (client_server_link PSi_PS_1_0_0 PermSi_PermS_1_0_0)
41        (client_server_link PSi_PS_1_0_0 Ai_Auth_1_0_0)
42        (client_server_link PSi_PS_1_0_0 USi_US_1_0_0)
43        (database_connection PSi_PS_1_0_0 DBPSi_DBPS_1_0_0)
44      )
45    )
46  )
```

Listing A.3 Configuration plan generated for creating GDE architecture with synchronous client-server calls

```
1  in parallel:
2    Action [actionName=create_databasesystem, arguments=[dbusi_dbus_1_0_0]]
3    Action [actionName=create_databasesystem, arguments=[dbpsi_dbps_1_0_0]]
4    Action [actionName=create_databasesystem, arguments=[dbfsi_dbfs_1_0_0]]
5    Action [actionName=create_databasesystem, arguments=[dbpermi_dbperm_1_0_0]]
6    Action [actionName=create_microservice, arguments=[permsi_perms_1_0_0]]
7    Action [actionName=create_microservice, arguments=[ai_auth_1_0_0]]
8    Action [actionName=create_microservice, arguments=[usi_us_1_0_0]]
9    Action [actionName=create_microservice, arguments=[fsi_fs_1_0_0]]
10   Action [actionName=create_microservice, arguments=[psi_ps_1_0_0]]
11 in parallel:
12   Action [actionName=link_client_server, arguments=[permsi_perms_1_0_0, ai_auth_1_0_0]]
13   Action [actionName=connect_microservice_to_databasesystem, arguments=[permsi_perms_1_0_0,
          dbpermi_dbperm_1_0_0]]
14   Action [actionName=link_client_server, arguments=[permsi_perms_1_0_0, usi_us_1_0_0]]
15   Action [actionName=link_client_server, arguments=[fsi_fs_1_0_0, ai_auth_1_0_0]]
16   Action [actionName=link_client_server, arguments=[fsi_fs_1_0_0, usi_us_1_0_0]]
17   Action [actionName=link_client_server, arguments=[fsi_fs_1_0_0, permsi_perms_1_0_0]]
18   Action [actionName=connect_microservice_to_databasesystem, arguments=[fsi_fs_1_0_0,
         dbfsi_dbfs_1_0_0]]
19   Action [actionName=link_client_server, arguments=[ai_auth_1_0_0, usi_us_1_0_0]]
20   Action [actionName=link_client_server, arguments=[usi_us_1_0_0, permsi_perms_1_0_0]]
21   Action [actionName=link_client_server, arguments=[usi_us_1_0_0, ai_auth_1_0_0]]
22   Action [actionName=connect_microservice_to_databasesystem, arguments=[usi_us_1_0_0,
         dbusi_dbus_1_0_0]]
23   Action [actionName=link_client_server, arguments=[psi_ps_1_0_0, fsi_fs_1_0_0]]
24   Action [actionName=link_client_server, arguments=[psi_ps_1_0_0, permsi_perms_1_0_0]]
25   Action [actionName=link_client_server, arguments=[psi_ps_1_0_0, ai_auth_1_0_0]]
26   Action [actionName=link_client_server, arguments=[psi_ps_1_0_0, usi_us_1_0_0]]
27   Action [actionName=connect_microservice_to_databasesystem, arguments=[psi_ps_1_0_0,
         dbpsi_dbps_1_0_0]]
```

Listing A.4 PDDL problem file for a minor revision with addition of logging in the GDE architecture

```
1  (define (problem microservice_planner_problem)
2    (:domain microservice_planner_domain)
3    (:objects
4      DBUSi_DBUS_1_0_0 - databasesystem
5      DBPSi_DBPS_1_0_0 - databasesystem
6      DBFSi_DBFS_1_0_0 - databasesystem
7      DBPERMi_DBPERM_1_0_0 - databasesystem
8      PermSi_PermS_1_0_0 - microservice
9      Ai_Auth_1_0_0 - microservice
10     FSi_FS_1_0_0 - microservice
11     USi_US_1_0_0 - microservice
12     PSi_PS_1_0_0 - microservice
13     LogSi_LS_1_0_0 - microservice
14     FSi_FS_1_1_0 - microservice
15     PubSubCi_PubSubConnector_1_0_0 - pubsubconnector
16   )
17   (:init
18     (databasesystem DBUSi_DBUS_1_0_0)
19     (databasesystem DBPSi_DBPS_1_0_0)
20     (databasesystem DBFSi_DBFS_1_0_0)
21     (databasesystem DBPERMi_DBPERM_1_0_0)
22     (microservice PermSi_PermS_1_0_0)
23     (client_server_link PermSi_PermS_1_0_0 Ai_Auth_1_0_0)
24     (client_server_link PermSi_PermS_1_0_0 USi_US_1_0_0)
25     (database_connection PermSi_PermS_1_0_0 DBPERMi_DBPERM_1_0_0)
26     (microservice Ai_Auth_1_0_0)
27     (client_server_link Ai_Auth_1_0_0 USi_US_1_0_0)
28     (microservice FSi_FS_1_0_0)
```

```
29      (client_server_link  FSi_FS_1_0_0 Ai_Auth_1_0_0)
30      (client_server_link  FSi_FS_1_0_0 USi_US_1_0_0)
31      (client_server_link  FSi_FS_1_0_0 PermSi_PermS_1_0_0)
32      (database_connection  FSi_FS_1_0_0 DBFSi_DBFS_1_0_0)
33      (microservice USi_US_1_0_0)
34      (client_server_link  USi_US_1_0_0 PermSi_PermS_1_0_0)
35      (client_server_link  USi_US_1_0_0 Ai_Auth_1_0_0)
36      (database_connection  USi_US_1_0_0 DBUSi_DBUS_1_0_0)
37      (microservice PSi_PS_1_0_0)
38      (client_server_link  PSi_PS_1_0_0 FSi_FS_1_0_0)
39      (client_server_link  PSi_PS_1_0_0 PermSi_PermS_1_0_0)
40      (client_server_link  PSi_PS_1_0_0 Ai_Auth_1_0_0)
41      (client_server_link  PSi_PS_1_0_0 USi_US_1_0_0)
42      (database_connection  PSi_PS_1_0_0 DBPSi_DBPS_1_0_0)
43    )
44    (:goal
45      (and
46        (databasesystem DBUSi_DBUS_1_0_0)
47        (databasesystem DBPSi_DBPS_1_0_0)
48        (databasesystem DBFSi_DBFS_1_0_0)
49        (databasesystem DBPERMi_DBPERM_1_0_0)
50        (microservice PermSi_PermS_1_0_0)
51        (microservice Ai_Auth_1_0_0)
52        (not (microservice FSi_FS_1_0_0))
53        (not (client_server_link  FSi_FS_1_0_0 Ai_Auth_1_0_0))
54        (not (client_server_link  FSi_FS_1_0_0 USi_US_1_0_0))
55        (not (client_server_link  FSi_FS_1_0_0 PermSi_PermS_1_0_0))
56        (not (database_connection  FSi_FS_1_0_0 DBFSi_DBFS_1_0_0))
57        (microservice USi_US_1_0_0)
58        (microservice PSi_PS_1_0_0)
59        (not   (client_server_link  PSi_PS_1_0_0 FSi_FS_1_0_0))
60        (client_server_link  PermSi_PermS_1_0_0 Ai_Auth_1_0_0)
61        (client_server_link  PermSi_PermS_1_0_0 USi_US_1_0_0)
62        (database_connection  PermSi_PermS_1_0_0 DBPERMi_DBPERM_1_0_0)
63        (client_server_link  Ai_Auth_1_0_0 USi_US_1_0_0)
64        (microservice LogSi_LS_1_0_0)
65        (client_server_link  USi_US_1_0_0 PermSi_PermS_1_0_0)
66        (client_server_link  USi_US_1_0_0 Ai_Auth_1_0_0)
67        (database_connection  USi_US_1_0_0 DBUSi_DBUS_1_0_0)
68        (client_server_link  PSi_PS_1_0_0 FSi_FS_1_1_0)
69        (client_server_link  PSi_PS_1_0_0 PermSi_PermS_1_0_0)
70        (client_server_link  PSi_PS_1_0_0 Ai_Auth_1_0_0)
71        (client_server_link  PSi_PS_1_0_0 USi_US_1_0_0)
72        (database_connection  PSi_PS_1_0_0 DBPSi_DBPS_1_0_0)
73        (microservice FSi_FS_1_1_0)
74        (client_server_link  FSi_FS_1_1_0 Ai_Auth_1_0_0)
75        (client_server_link  FSi_FS_1_1_0 USi_US_1_0_0)
76        (client_server_link  FSi_FS_1_1_0 PermSi_PermS_1_0_0)
77        (database_connection  FSi_FS_1_1_0 DBFSi_DBFS_1_0_0)
78        (pubsubconnector PubSubCi_PubSubConnector_1_0_0)
79        (producer_to_connector_link  FSi_FS_1_1_0 PubSubCi_PubSubConnector_1_0_0)
80        (consumer_to_connector_link  LogSi_LS_1_0_0 PubSubCi_PubSubConnector_1_0_0)
81      )
82    )
83  )
```

Listing A.5 Configuration plan generated for the corresponding minor revision of the GDE architecture

```
1 in parallel:
2   Action [actionName=create_pubsubconnector, arguments=[pubsubci_pubsubconnector_1_0_0]]
3 in parallel:
4   Action [actionName=unlink_client_server, arguments=[fsi_fs_1_0_0, ai_auth_1_0_0]]
```

```
5     Action [actionName=disconnect_microservice_from_databasesystem, arguments=[fsi_fs_1_0_0,
          dbfsi_dbfs_1_0_0]]
6     Action [actionName=unlink_client_server, arguments=[fsi_fs_1_0_0, usi_us_1_0_0]]
7     Action [actionName=unlink_client_server, arguments=[fsi_fs_1_0_0, permsi_perms_1_0_0]]
8     Action [actionName=unlink_client_server, arguments=[psi_ps_1_0_0, fsi_fs_1_0_0]]
9     Action [actionName=create_microservice, arguments=[fsi_fs_1_1_0]]
10    Action [actionName=create_microservice, arguments=[logsi_ls_1_0_0]]
11  in parallel:
12    Action [actionName=remove_microservice, arguments=[fsi_fs_1_0_0]]
13    Action [actionName=link_client_server, arguments=[psi_ps_1_0_0, fsi_fs_1_1_0]]
14    Action [actionName=link_client_server, arguments=[fsi_fs_1_1_0, ai_auth_1_0_0]]
15    Action [actionName=link_client_server, arguments=[fsi_fs_1_1_0, usi_us_1_0_0]]
16    Action [actionName=link_client_server, arguments=[fsi_fs_1_1_0, permsi_perms_1_0_0]]
17    Action [actionName=connect_microservice_to_databasesystem, arguments=[fsi_fs_1_1_0,
          dbfsi_dbfs_1_0_0]]
18    Action [actionName=link_producer_to_pubsubconnector, arguments=[fsi_fs_1_1_0,
          pubsubci_pubsubconnector_1_0_0]]
19    Action [actionName=link_consumer_to_pubsubconnector, arguments=[logsi_ls_1_0_0,
          pubsubci_pubsubconnector_1_0_0]]
```

A.2 Architecture of Executor in Autonomic Manager for MIMOSAE

Figure A.1 presents a communication diagram that represents the general functioning of executor in our prototype. The implementation and documentation can be found at the URL: https://gitlabev.imtbs-tsp.eu/mimosae/mimosae/-/tree/main/Code/AutonomicManager/Executor.

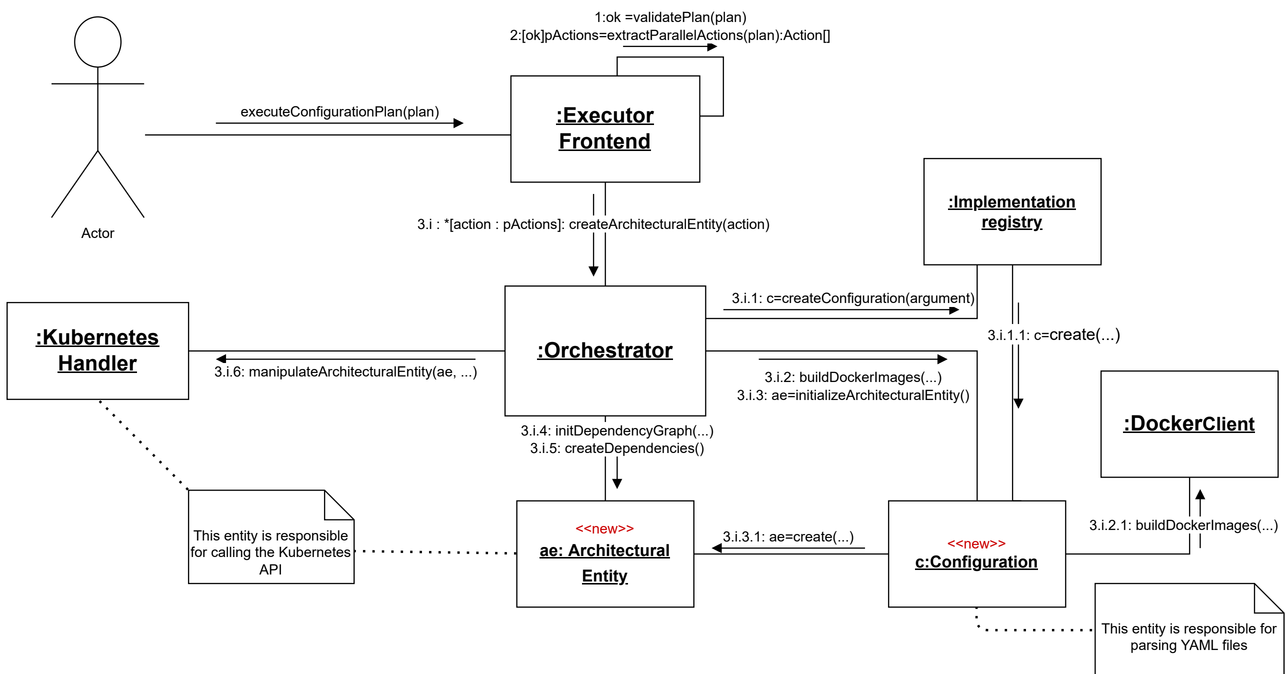


Figure A.1 – Communication diagram of executor's general functioning

A.3 Overview of the proposed Software Evolution Process

Figure A.2 represents an overview of our proposed software evolution process. At design time, the developer teams provide different versions of microservices in the code base, and then the architects build and manipulate the models for microservice architectures. These models are built at design time, and will be used at runtime. Our process is considered semi-automatic with a MAPE-K control loop at runtime. Semi-automatic means that one part of the control process is realized manually by engineers and another part is performed automatically by the autonomic manager. Here, we can see that we ignore the monitoring and analyzing steps, and instead get the necessary information about the version changes of microservices from the design time, that is our runtime models act as the knowledge base in the control loop. The architects manually provide a source configuration and a target configuration of the microservice architecture, so that the planning can compute a configuration plan of the version changes, and this reconfiguration plan can be then executed and applied to the running application.

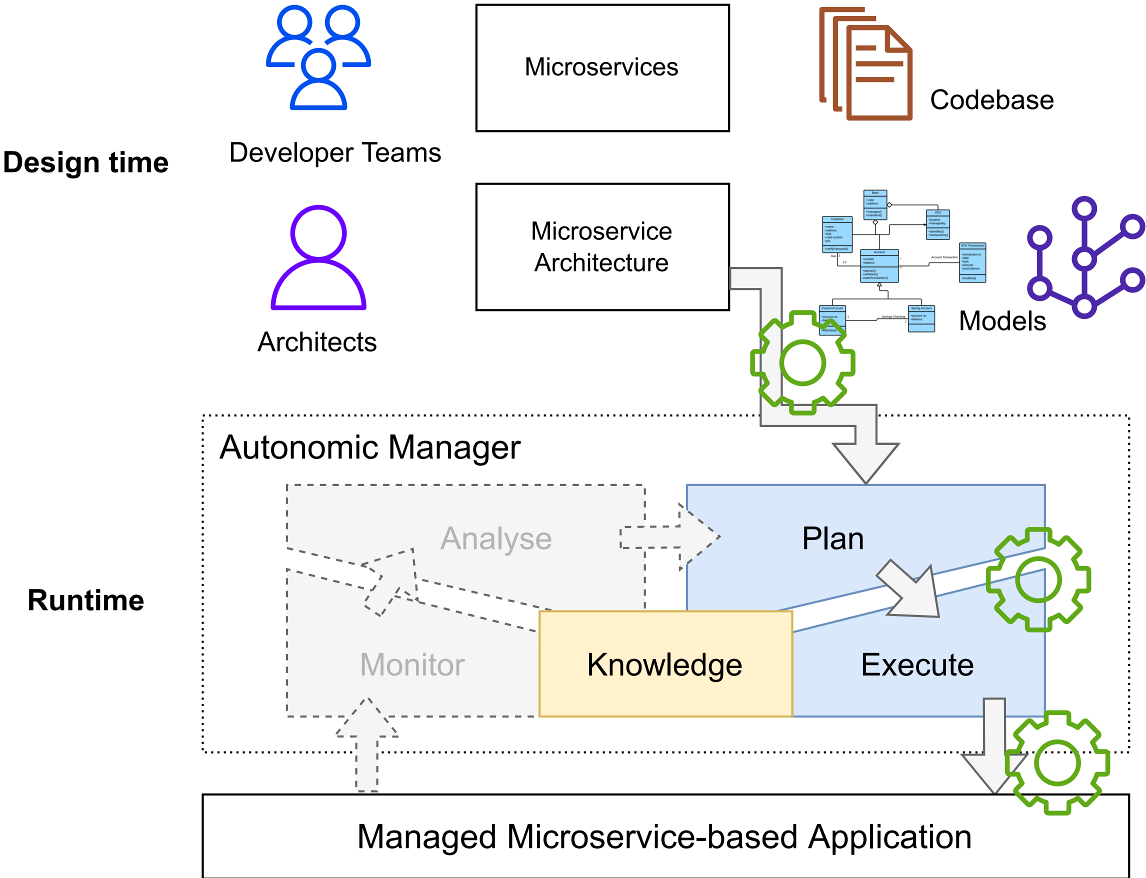


Figure A.2 – Overview of software evolution process

ECOLE
DOCTORALE

Titre : Évolution des Applications à base de Microservices : Modélisation et Mise à jour dynamique correcte

Mots clés : Architecture à base de microservices ; Évolution des logiciels ; Gestion des versions ; Modèle à l'exécution ; Mise à jour dynamique des logiciels ; Détection des conditions de mise à jour basée sur des instantanés.

Résumé : Les architectures à base de microservices permettent de construire des systèmes répartis complexes composés de microservices indépendants. Le découplage et la modularité des microservices facilitent leur remplacement et leur mise à jour de manière indépendante. Depuis l'émergence du développement agile et de l'intégration continue (*DevOps* et *CI/CD*), la tendance est aux changements de version plus fréquents et en cours d'exécution des applications. La réalisation des changements de version est effectuée par un processus d'évolution consistant à passer de la version actuelle de l'application à une nouvelle version. Cependant, les coûts de maintenance et d'évolution de ces systèmes répartis augmentent rapidement avec le nombre de microservices.
L'objectif de cette thèse est de répondre aux questions suivantes : Comment aider les ingénieurs à mettre en place une gestion de version unifiée et efficace pour les microservices et comment tracer les changements de version dans les applications à base de microservices ? Quand les applications à base de microservices, en particulier celles dont les activités sont longues, peuvent-elles être mises à jour dynamiquement sans arrêter l'exécution de l'ensemble du système ? Comment la mise à jour doit-elle être effectuée pour assurer la continuité du service et maintenir la cohérence du système ?
En réponse à ces questions, cette thèse propose deux contributions principales. La première contribution est constituée de modèles architecturaux et d'un graphe d'évolution pour modéliser et tracer la gestion des versions des microservices. Ces modèles sont construits lors de la conception et utilisés durant l'exécution. Cette contribution aide les ingénieurs à abstraire l'évolution architecturale afin de gérer les déploiements lors d'une reconfiguration, et fournit la base de connaissances nécessaire à un intergiciel de gestion autonomique des activités d'évolution. La deuxième contribution est une approche basée sur les instantanés pour la mise à jour dynamique (DSU) des applications à base de microservices. Les instantanés répartis cohérents de l'application en cours d'exécution sont construits pour être utilisés dans la spécification la continuité du service, l'évaluation des conditions de mise à jour sûre et dans la mise en œuvre des stratégies de mise à jour. La complexité en nombre de messages de l'algorithme DSU n'est alors pas égale à la complexité de l'application répartie, mais correspond à la complexité de l'algorithme de contruction d'un instantané réparti cohérent.

Title : Evolution of Microservice-based Applications : Modelling and Safe Dynamic Updating

Keywords : Microservice architecture ; Software evolution ; Version management ; Model at runtime ; Dynamic software updating ; Snapshot-based update condition detection.

Abstract : Microservice architectures contribute to building complex distributed systems as sets of independent microservices. The decoupling and modularity of distributed microservices facilitates their independent replacement and upgradeability. Since the emergence of agile DevOps and CI/CD, there is a trend towards more frequent and rapid evolutionary changes of the running microservice-based applications in response to various evolution requirements. Applying changes to microservice architectures is performed by an evolution process of moving from the current application version to a new version. The maintenance and evolution costs of these distributed systems increase rapidly with the number of microservices.
The objective of this thesis is to address the following issues : How to help engineers to build a unified and efficient version management for microservices and how to trace changes in microservice-based applications ? When can microservice-based applications, especially those with long-running activities, be dynamically updated without stopping the execution of the whole system ? How should the safe updating be performed to ensure service continuity and maintain system consistency ?
In response to these questions, this thesis proposes two main contributions. The first contribution is runtime models and an evolution graph for modelling and tracing version management of microservices. These models are built at design time and used at runtime. It helps engineers abstract architectural evolution in order to manage reconfiguration deployments, and it provides the knowledge base to be manipulated by an autonomic manager middleware in various evolution activities. The second contribution is a snapshot-based approach for dynamic software updating (DSU) of microservices. The consistent distributed snapshots of microservice-based applications are constructed to be used for specifying continuity of service, evaluating the safe update conditions and realising the update strategies. The message complexity of the DSU algorithm is not the message complexity of the distributed application, but the complexity of the consistent distributed snapshot algorithm.

Institut Polytechnique de Paris
91120 Palaiseau, France