UNIVERSITÉ DE SHERBROOKE
Faculté de génie
Département de génie électrique et de génie informatique

Patrons de conception pour l'intégration graduelle de mécanismes d'adaptation dans les interfaces graphiques

Mémoire de maîtrise
Spécialité : génie électrique

Samuel LONGCHAMPS

Jury : Ruben GONZALEZ-RUBIO (directeur)
Philippe MABILLEAU
Frédéric MAILHOT (rapporteur)

Sherbrooke (Québec) Canada Août 2017

RÉSUMÉ

Les interfaces graphiques de logiciels modernes requièrent de plus en plus de s'adapter à diverses situations et divers utilisateurs, rendant leur développement plus complexe. Peu de guides et de solutions à faible coût d'intégration dans un projet existent et sont trop souvent dépendants d'une technologie ou d'une plateforme donnée.

Ce mémoire présente une technique pour l'implémentation graduelle de comportements adaptatifs dans les interfaces graphiques par le biais de patrons de conception. Les patrons de conception sont des solutions formalisées répondant à des problèmes récurrents, dans ce cas-ci de structuration d'un logiciel pour l'ajout de l'adaptation. Ces derniers sont présentés dans un format normalisé et une implémentation de référence a été développée sous forme de librairie baptisée AdaptivePy. Un prototype démonstratif est utilisé pour comparer une approche d'implémentation *ad hoc* à celle utilisant la librairie et donc les patrons. Les gains observés sont au niveau de la séparation des préoccupations, de la cohésion des méthodes, de la localisation des changements pour l'ajout de l'adaptation et de l'extensibilité.

Aussi, ce mémoire présente des métriques visant la vérification de l'organisation des composants d'un logiciel structuré par l'application des patrons de conception. Ces métriques sont des indicateurs de la proportion des situations contextuelles du système que supporte un composant. Les métriques et leur calcul sont présentés dans un format basé sur celui de l'ISO/IEC 25023 et une implémentation de référence a également été développée. Une application typique est évaluée grâce aux métriques et des actions correctives sont présentées pour résoudre les problèmes détectés. L'utilité des métriques pour valider une application développée en utilisant la structure induite par les patrons de conception est ainsi mise en évidence.

La méthodologie du projet a suivi un processus itératif pour l'élaboration des patrons de conception et la recherche des métriques pour appuyer leur application dans un contexte pratique. Par l'analyse de la littérature pour identifier les concepts communs de l'adaptation et les éléments de mesure dans le domaine de l'adaptation, des solutions plus générales et adaptées à une grande variété de domaines d'application sont proposées. Parmi les contributions du projet sont trois nouveaux patrons de conceptions : *Moniteur*, *Proxy routeur* et *Composant adaptatif*. Aussi, deux métriques spécifiques ont été formalisées : la *couverture modélisée de l'espace d'adaptation* ainsi que la *couverture effective de l'espace d'adaptation*. En plus, deux métriques générales supplémentaires sont proposées : la *profondeur de l'arbre de substitution* et la *stabilité de l'adaptation*.

Mots-clés : adaptation, patron, conception, logiciel, composant, interface graphique, métrique, validation & vérification

TABLE DES MATIÈRES

LISTE DES FIGURES

LISTE DES TABLEAUX

LEXIQUE

Adaptation logicielle L'adaptation logicielle est le principe selon lequel un logiciel parvient à changer son comportement et/ou sa structure au moment de son exécution de façon autonome en réponse à des stimuli provenant de son environnement (externe) ou de son propre état (interne).

***Ad hoc*, Solution** *Ad hoc* vient du latin et se traduit par "pour ceci". Une solution est dite *ad hoc* lorsqu'elle est spécifique au système s'y rattachant et n'est pas généralisable.

***Binding*, Librairie de** Librairie effectuant un pont vers une librairie implémentée dans un autre langage. Dans le cas de Qt qui est une librairie C++, PyQt est un binding fonctionnellement équivalent avec comme cible le langage Python.

Commit Transaction avec un système de versionnement de code qui représente la soumission d'une nouvelle version du code.

Framework Suite intégrée d'outils logiciels implémentant diverses fonctionnalités agissant à titre de base pour une application.

Middleware Logiciel offrant un support de fonctionnalités entre le bas et le haut niveau. Dans ce cas, le bas niveau correspond au système d'exploitation et le haut niveau aux applications.

Modèle conceptuel Organisation des concepts d'un champ de recherche sous forme d'un modèle.

Patron de conception Une solution éprouvée à un problème récurrent. Dans le cadre d'un logiciel, cette solution est une façon de faire pour régler un problème qui mène à une bonne qualité logicielle.

Préoccupations, Séparation des (*separation of concerns*) Une préoccupation dans le cadre d'un logiciel est un concept auquel sont liées différentes fonctionnalités. La séparation des préoccupations vise à découpler les fonctionnalités en fonction de leur préoccupation, de sorte que chaque groupement de fonctionnalités soit lié à une seule préoccupation.

Qualité de Service La qualité de service est est un indicateur visant à quantifier l'atteinte de requis de qualité pour une préoccupation, typiquement dans le domaine des télécommunications, mais pouvant s'étendre à d'autres domaines. Un exemple est le délai de réponse d'un serveur ou la latence : la qualité de service est ici la meilleure lorsqu'elle approche 0.

Toolkit Librairie de composants personnalisables destinée à la conception d'interfaces graphiques.

Widget Composant dans le contexte d'un *toolkit*.

LISTE DES ACRONYMES

CASC *Composite Adaptation Space Coverage* (couverture composite de l'espace d'adaptation)

EASC *Effective Adaptation Space Coverage* (couverture de l'espace d'adaptation effective)

MASC *Modeled Adaptation Space Coverage* (couverture de l'espace d'adaptation modélisé)

PASC *Primitive Adaptation Space Coverage* (couverture primitive de l'espace d'adaptation)

POO Programmation Orientée-Objet

PVP *Parameter Value Provider* (fournisseur de valeur de paramètre)

QdS Qualité de Service

QME *Quality Measure Element* Élément de mesure de la qualité

RSC *Recurrent State Space Coverage* (couverture récurrente de l'espace d'états)

SC *State space Coverage* (couverture de l'espace d'états)

V&V Vérification et Validation

CHAPITRE 1

Introduction

1.1 Mise en contexte et problématique

Depuis plus d'une décennie, les logiciels ont envahi les objets de notre quotidien. L'avènement de l'Internet des Objets (*Internet of Things*), soit l'intégration de microprocesseurs et d'inter-connectivité dans des appareils traditionnels comme des thermostats et des lampes, a grandement poussé cette vague. De plus en plus, les appareils utilisés dans la vie de tous les jours sont contrôlés et dépendent de logiciels. Ce marché émergeant vise à personnaliser ces différents appareils aux préférences et contextes particuliers de chacun des foyers et des utilisateurs.

Cette idée de personnaliser l'expérience des utilisateurs avec leurs appareils n'est pas nouvelle. En effet, il y a déjà plusieurs décennies que le domaine de l'adaptation des logiciels se développe. La vision plus générale est de faire en sorte qu'un système puisse changer son comportement et sa structure interne pour satisfaire à certains critères. Par exemple, un système rencontrant une erreur pourrait prendre des mesures et changer le module fautif pour un autre. Ce principe est appelé « autoguérison » (*self-healing*) [17]. Plus près de l'utilisateur, un logiciel pourrait personnaliser son interface en fonction du type d'utilisateur ou de son environnement. Un exemple souvent rencontré dans les logiciels existants est l'utilisation de différents types d'interfaces, l'une basique et l'autre avancée. Le choix de l'interface est fait en fonction des caractéristiques de l'utilisateur comme des problèmes moteurs spécifiques ou simplement son degré de familiarité avec le logiciel. Dans ce cas, on parle plutôt d'adaptation basée sur la modélisation des utilisateurs [34].

Beaucoup de chercheurs se sont penchés sur les techniques et mécanismes permettant l'atteinte de buts précis par l'adaptation. Cependant, l'intégration de ces mécanismes dans l'ensemble des logiciels reste difficile à faire puisque beaucoup proposent leur propre solution d'intégration, généralement sous forme de *framework*. En plus, plusieurs solutions proposées à ce jour sont directement liées à un domaine d'application précis, typiquement les applications distribuées en réseau. Dans ce domaine, les requis sont souvent rattachés à la qualité de service (QdS) de serveurs et de services. Ainsi, les mécanismes proposés sont difficiles à mettre en place dans d'autres logiciels. Par exemple, les implémentations

dans le domaine des interfaces graphiques restent relativement peu explorées et la QdS est interprétée différemment.

Le problème central semble être l'absence de techniques générales pour la mise en place de l'adaptation dans l'ensemble des logiciels et plus particulièrement les logiciels existants. Comme les logiciels deviennent plus répandus et utilisés dans une variété de contextes différents, l'adaptation est devenue requise par nécessité. Il est en effet plus efficace d'adapter une solution générale à des domaines spécifiques plutôt que de développer des solutions complètes spécifiques à chacun des domaines. Cependant, en l'absence de techniques générales, il reste difficile de mettre en place des mécanismes d'adaptation graduellement et d'une façon qui assure la maintenabilité. En plus, comme les logiciels adaptatifs sont différents des logiciels traditionnels par leur organisation déterminée de façon dynamique, les techniques de vérification et de validation (V&V) existantes sont insuffisantes pour déterminer si une implémentation n'entrera pas dans un état où son comportement est indéfini.

1.2 Définition du projet de recherche

Le présent projet de recherche propose une technique suffisamment générique pour permettre l'application d'une grande variété de mécanismes d'adaptation dans les logiciels et qui reste neutre au niveau des technologies utilisées. Il est important de mettre l'accent sur le but qui est de proposer une façon de mettre en place une *structure de base* dont les modules réalisent les concepts nécessaires à l'adaptation au sens large. Dans le contexte du présent projet, le domaine des interfaces graphiques a été choisi comme cible afin de réduire le cadre de la recherche. Ce domaine a été choisi puisqu'il est de plus en plus soumis à des requis d'adaptation vu l'hétérogénéité des appareils. De plus, la structure typiquement par composants des *toolkits* est largement répandue dans le domaine du logiciel.

Au total, trois patrons de conception sont proposés pour mettre en place une structure de base pour introduire de l'adaptation dans un logiciel. La schématisation sous forme de patrons de conception est la technique choisie puisqu'elle permet de communiquer des solutions dans un format clair et à un niveau de détail adéquat. Ces schémas montrent l'organisation d'un groupe de modules ayant des rôles précis et qui, utilisés ensemble, offrent une solution à un problème récurrent en logiciel. Dans notre cas, le problème global est l'ajout d'un mécanisme d'adaptation à une application. Plus précisément, deux des patrons présentés répondent à un problème en lien avec une préoccupation de l'adaptation. Le troisième lie plutôt les deux premiers pour former une structure propice à l'adaptation.

Comme il est également nécessaire d'assurer que la mise en place de l'adaptation est structurellement adéquate, le présent projet inclut la proposition d'une technique de vérification basée sur la structure de base établie. Une technique de V&V répandue est l'utilisation de métriques de qualité pour évaluer la qualité de façon quantitative. Une métrique offre un indicateur permettant de qualifier le degré de qualité d'une implémentation en fonction d'une préoccupation particulière, par exemple la complexité d'une fonction. Pour ce faire, une métrique permet le calcul d'une valeur et le degré de qualité peut être interprété en comparant cette valeurs à des valeurs bornes (minimales et maximales). Une « bonne » valeur dépend de la métrique et il faut généralement minimiser ou maximiser la valeur vers l'une des bornes pour améliorer la qualité. Le présent projet inclut la proposition d'une nouvelle métrique spécifique aux logiciels adaptatifs. Cette métrique fournit un indicateur de la proportion des états du système qui sont supportés par la structure des composants depuis lesquels ce dernier est formé.

Les gains attendus par l'application de techniques générales pour la mise en place de l'adaptation dans les logiciels sont une meilleure qualité des implémentations et une application des mécanismes d'adaptation dans une plus grande variété de logiciels. Plus particulièrement, l'utilisation de telles techniques devrait minimiser l'effort requis pour l'ajout de comportements adaptatifs dans des projets qui n'en comportent initialement pas. De plus, il devrait être plus facile de comparer différents mécanismes d'adaptation ainsi que différentes implémentations d'un même mécanisme. Dans le contexte des interfaces graphiques, il est attendu que la structure rend la conception et l'implémentation d'interfaces adaptatives plus graduelles et conserve une approche similaire aux techniques existantes dans ce domaine.

1.3 Objectifs du projet de recherche

L'objectif principal de la recherche est la création de nouveaux patrons de conception qui représentent des solutions suffisamment générales à des problèmes liés aux préoccupations spécifiques de l'adaptation. De façon plus concise, on cherche à savoir et à montrer : *Quels patrons de conception permettent l'implémentation graduelle de mécanismes d'adaptation dans les interfaces graphiques ?* Pour atteindre cet objectif principal, il est nécessaire d'atteindre des objectifs secondaires. Ces derniers sont :

Proposition de patrons de conceptions

1. Relever de la littérature une banque de patrons de conception liés au domaine de l'adaptation logicielle
2. Analyser les patrons et formaliser les concepts fondamentaux solutionnant les problèmes communs de l'adaptation logicielle
3. Déterminer les lacunes et avantages des patrons de conceptions trouvés
4. Proposer des patrons de conceptions liant les concepts fondamentaux et adressant les lacunes trouvées dans ceux déjà existants
5. Implémenter les patrons de conceptions afin de valider leur fonctionnement dans une application concrète
6. Comparer une solution *ad hoc* à une nouvelle utilisant les patrons de conception proposés

De nouveaux patrons de conception permettront de traduire les éléments récurrents qui caractérisent une implémentation de qualité dans le but d'être utilisés dans divers logiciels. Chaque patron aura comme préoccupation un aspect de l'adaptation qui favorisera une meilleure qualité logicielle.

Proposition de métriques

1. Relever de la littérature une banque de métriques liées à la qualité des logiciels adaptatifs, particulièrement au niveau de la structure
2. Analyser les éléments liés aux métriques existantes et les pistes proposées par les recherches existantes
3. Extraire depuis la structure de base proposée des éléments de mesure permettant de quantifier la qualité
4. Proposer des métriques utilisant des éléments de mesure et qui quantifient un aspect de la qualité structurelle des logiciels adaptatifs
5. Implémenter le calcul des métriques et valider son utilisation avec des éléments de mesure tirés d'une application concrète
6. Analyser les résultats des métriques en variant les éléments de mesure dans l'application concrète et proposer des actions correctives

La proposition de la métrique s'inscrit dans l'objectif de proposer une solution *éprouvée*. En effet, il est nécessaire d'effectuer la vérification que le logiciel où les patrons de conception ont été utilisés satisfait des requis de qualité hors de l'atteinte des patrons seuls. Ainsi, la métrique proposée a comme but d'identifier les implémentations qui aboutissent en un système contenant des problèmes.

1.4 Contributions originales

Plusieurs contributions originales sont issues de ce projet. Les trois patrons de conception proposés sont le *Moniteur*, *Proxy routeur* et *Composant adaptatif*.

- Le patron *Moniteur*[1] vise la modélisation, l'acquisition et la distribution de données pour l'adaptation
- Le patron *Proxy routeur* vise la réorganisation des chemins d'appels entre différents composants
- Le patron *Composant adaptatif* vise la composition d'architectures et de hiérarchies de composants qui ont un comportement adaptatif reposant autant sur la substitution que la paramétrisation

Les métriques pour la vérification de logiciels adaptatifs sont la couverture de l'espace d'adaptation modélisée et la couverture de l'espace d'adaptation effective. En plus, des pistes d'analyse et des actions correctives sont proposées pour accompagner les développeurs dans l'utilisation de la métrique.

Pour mettre en pratique les différents patrons et métriques, des développements logiciels ont été réalisés et rendus disponibles sous licence libre du Québec (LiLiQ-P v1.1). Ces derniers sont disponibles aux adresses suivantes :

- **AdaptivePy :** `https://gitlab.com/memophysic/adaptive_py_lib`
- **Métriques :** `https://gitlab.com/memophysic/adaptive_py_metrics`

AdaptivePy est également disponible sur PyPi, le répertoire des librairies Python, sous le nom `adaptivepy`[2].

[1] Ne pas confondre avec le patron Moniteur utilisé dans le domaine de la concurrence

[2] La librairie peut être installée grâce à l'outil pip avec la commande `pip3 install adaptivepy` sur une distribution GNU/Linux

1.5 Plan du document

Ce document est découpé en différents chapitres. Le chapitre 2 présente l'état de l'art dans le domaine de l'adaptation dans les logiciels et illustre les travaux sur lesquels la présente recherche est basée. Le chapitre 3 introduit un premier article qui a été présenté dans le cadre de la conférence *Adaptive 2017* et explique en plus grands détails la méthodologie et les développements qui ont permis la proposition des trois patrons de conception. Ce premier article est reproduit au chapitre 4. Un second article est introduit au chapitre 5 et les démarches pour l'élaboration des métriques et des solutions pour la vérification de logiciels adaptatifs y sont présentées. Le second article est reproduit au chapitre 6. Une discussion sur les contributions du projet de recherche et leurs implications est le sujet du chapitre 7. Ce chapitre présente également des idées qui n'ont pas été couvertes dans les articles, mais qui ont tout de même fait partie de la recherche. Le mémoire se conclut avec le chapitre 8 où un sommaire des travaux est présenté ainsi que diverses propositions pour des travaux futurs.

CHAPITRE 2

État de l'art

Le présent chapitre a pour objectif de présenter un état de l'art du domaine de l'adaptation dans les logiciels avec une attention particulière portée sur les méthodes d'intégration de l'adaptation. Un aperçu du champ de recherche est d'abord présenté, suivi d'une mise en contexte de l'adaptation par rapport au dynamisme dans les logiciels. Ensuite, différents types de modèles de composants adaptatifs sont présentés. Les patrons de conception connus qui se rapprochent du but du présent projet de recherche sont présentés par la suite ainsi qu'un résumé des aspects de la V&V des logiciels adaptatifs. Finalement, des résumés des enquêtes les plus récentes dans le domaine de l'adaptation dans les logiciels sont présentés, suivis d'une mise en évidence des éléments de recherche qui sont représentés dans le présent projet.

À noter que les parties « champ de recherche », « dynamisme et adaptation », et « patrons de conception » sont en grande partie tirées du travail réalisé lors de la définition de projet en préparation à l'écriture de ce mémoire.

2.1 Champ de recherche

La discipline de l'adaptation logicielle est issue de divers besoins et requis auxquels font face les systèmes informatiques modernes. Un logiciel adaptatif est généralement « un logiciel pouvant changer son comportement à l'exécution. » [7] Traditionnellement, le cycle de développement des logiciels est constitué de différentes phases comme la conception, l'implémentation, la validation, le support, etc. Ces phases sont aujourd'hui souvent effectuées de façon itérative et le logiciel est livré de façon continue. Lorsque le logiciel compte un nombre de fonctionnalités clés qui en font une solution complète, ce dernier est livré sous forme de version finale. Si aucune autre fonctionnalité n'est requise, la phase finale de support vise ensuite à corriger les bogues et problèmes survenant dans l'exécution du système.

Actuellement, il devient nécessaire pour les logiciels d'être autonomes, c'est-à-dire que leur cycle de développement ne se limite plus au procédé itératif, mais bien qu'il continue après la livraison de façon autonome [43]. Cette autonomie permet aux logiciels de s'adapter à des situations qui auraient normalement requis une intervention humaine. Un exemple de

ce type d'adaptation est la capacité pour un système composé de plusieurs serveurs de modifier la façon dont ces derniers traitent les requêtes clientes pour garantir un délai de réponse maximal[1]. Normalement, une intervention humaine aurait été nécessaire pour configurer le système dès qu'il aurait été détecté que le temps de réponse des serveurs était trop long. Dans le cas des logiciels adaptatifs, ils sont en mesure d'acquérir le temps de réponse et de modifier leur configuration pour atteindre certaines cibles définies par les concepteurs de façon autonome. C'est donc dire que le logiciel effectue lui-même une partie de la configuration et même du développement après avoir été livré.

Évidemment, il est nécessaire de trouver des techniques pour faire l'implémentation de différents mécanismes d'adaptation. L'adaptation dans les logiciels est alors la recherche de ces techniques et de façons de valider que les objectifs stipulés pour un système sont atteints. Le présent projet de recherche vise plus particulièrement la façon de structurer les applications pour faciliter l'ajout de mécanismes d'adaptation en maximisant une séparation des préoccupations.

2.2 Dynamisme et adaptation

Au travers des années, les outils de conception de logiciels ont connu une sophistication qui a apporté plusieurs fonctionnalités autrefois impossible. Par exemple, la compilation permet d'abstraire différentes spécificités liées aux architectures des processeurs et aux systèmes d'exploitation, permettant l'écriture d'une application logicielle dans un langage unifié. Cette flexibilité accrue a également permis d'ajouter de plus en plus de comportements dynamiques dans les applications. Ce type de fonctionnalité est appelé *dynamisme* puisqu'il réfère à la capacité de résoudre un ou plusieurs niveaux d'indirection par un procédé automatisé. L'indirection, qui est le principe d'utiliser une référence vers une entité plutôt que cette dernière directement, est par exemple utilisé dans le cas de la compilation pour résoudre vers quelle procédure concrète un appel de procédure est redirigé en fonction des types de structures utilisés (acheminement dynamique ou *dynamic dispatch*). Ce principe est l'un des éléments fondamentaux de la programmation orientée-objet (POO), qui apporte un niveau d'indirection non seulement aux procédures, mais également aux structures de données sous forme de classes.

La dynamisation des logiciels induit une complexité inhérente à ces derniers et il devient alors important de développer des méthodes et outils pour effectuer leur V&V. Dans tous

[1]Un délai maximal garanti implique qu'une requête ne peut prendre plus que t unité de temps à être traitée.

les cas, il s'agit alors de s'assurer que le logiciel répond à certaines exigences (ex. : performance, taille en mémoire, etc.), que sa structure est valide (ex. : lexeur-parseur) et son comportement est bel et bien celui qui est attendu (ex. : tests unitaires et d'intégration). Des requis de validation plus récents sont d'assurer que le logiciel ne peut être corrompu (ex. : exécuter des procédures introduites de façon malveillante) et que lorsqu'un problème survient, ce dernier peut rétablir le service qu'il offre ou se tourner vers une solution de rechange [17]. Ces derniers requis sont plus difficiles à valider puisqu'ils sont globaux au logiciel et leurs solutions sont excessivement dynamiques (sont liés à plusieurs niveaux d'indirection). L'adaptation logicielle fait partie de ces solutions puisqu'elle vise à dynamiser jusqu'à l'architecture même du logiciel, par exemple en remplaçant un ou plusieurs composants offrant un certain service par un autre de rechange ou en modifiant les algorithmes utilisés à l'intérieur de ce dernier. Les recherches dans le domaine ont initialement visé le développement de techniques pour implémenter de tels mécanismes et se sont dirigées, dans les dernières années, vers l'élaboration de stratégies d'adaptation plus spécifiques à certains types de problèmes et de systèmes.

Une architecture logicielle populaire, particulièrement dans les systèmes distribués, est l'architecture par composants. Elle constitue également l'architecture à l'étude pour le présent projet de recherche, à la différence que l'accent n'est pas sur les systèmes distribués, mais plutôt sur tout type de logiciel développé avec la POO. La programmation basée sur les interfaces (*interface-based programming*) est la technique utilisée pour spécifier les composants et réaliser leurs connexions dans le cadre du présent projet. Afin de mieux cibler le projet, les interfaces graphiques ont été choisies comme une étude de cas puisqu'elles sont typiquement réalisées avec des composants dans un contexte de POO.

2.3 Modèles de composant adaptatif

La programmation orientée composant vise à faire la conception d'une hiérarchie de composants simples qui ensemble forment un composant complexe. Cette approche a gagné beaucoup en popularité depuis les années 90, entre autres avec l'arrivée de modèles comme le *System Object Model* de IBM, le *Component Object Model* de Microsoft et *Common Object Request Broker Architecture (CORBA)* du Object Management Group. Ces derniers ont été développés pour ajouter un niveau d'indirection aux composants et ainsi faciliter l'interopérabilité entre des composants développés en différents langages. Dans le cas d'un logiciel donné, les composants servent à mettre en place une séparation des préoccupations explicites de sorte que différents composants offrant un certain service peuvent être assemblés pour fournir un service plus complexe et donc une solution à un problème de

plus haut niveau. Effectuer la conception de composants pouvant s'adapter et créer des composants dont la préoccupation serait uniquement l'adaptation ont alors commencé à intéresser les chercheurs.

2.3.1 Modèle pour architecture distribuée

Taylor *et al.* [45] ont proposé en 1996 un modèle générique de composant hautement découplé servant de base à l'élaboration d'un marché des composants. Le but était alors de supporter les systèmes du futur qui seraient « distribués, complexes, multimédias, hétérogènes et multi-utilisateurs ». Ces caractéristiques sont les mêmes qui ont poussé le développement de l'adaptation logicielle, particulièrement lorsqu'il s'agit d'applications distribuées comme des réseaux de serveurs. Une qualité intéressante de ce modèle est l'utilisation de messages dans un langage intermédiaire pour la communication entre tous les composants. De cette façon, chaque composant peut être distribué dans son propre processus et sur une machine distincte. Cependant, les mécanismes nécessaires afin de mettre en place l'adaptation comme la détection de changements et le remplacement de composants ne faisaient pas partie du modèle : l'architecture restait essentiellement statique.

2.3.2 Composant adaptatif par activation/désactivation

Chen *et al.* [7] proposent en 2001 un modèle de composants pour mettre en place l'adaptation dans un logiciel distribué (voir modèle d'un *Adaptive Component* (AC), figure 2.1). Ils identifient certains défis comme effectuer la coordination entre différentes machines afin de ne pas suspendre le système lorsque l'adaptation est mise en place. Ils décrivent deux types de sous-composants qui forment un « composant adaptatif » : un module d'adaptation

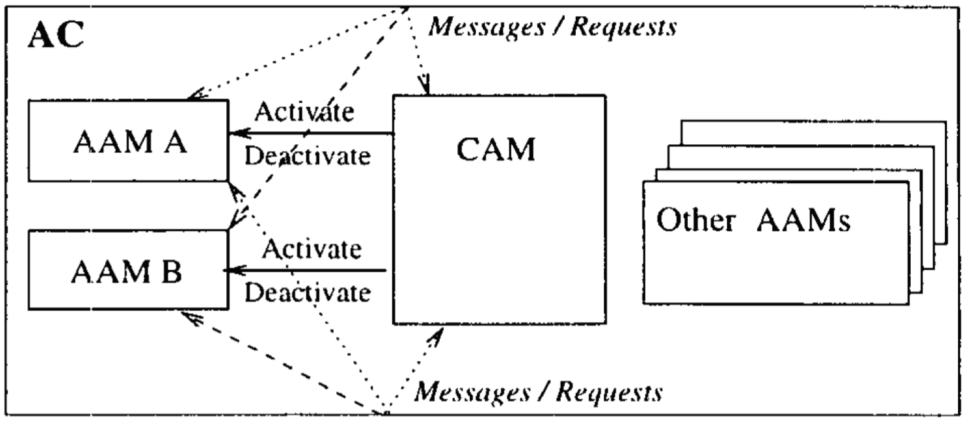

Figure 2.1 Composant adaptatif de Chen *et al.* (tiré de [7])

du composant ou *component adaptor module (CAM)* ainsi qu'un module d'algorithme conscient de l'adaptation ou *adaptation-aware algorithm module (AAM)*. Chaque CAM peut activer ou désactiver un AAM et les requêtes sont envoyées uniquement à ceux qui sont activés. Le processus d'adaptation est d'abord de détecter un changement dans l'environnement d'exécution, de décider des AMMs à activer et désactiver communautairement et d'effectuer les actions d'adaptation en assurant qu'il n'y aura pas d'arrêt des services en cours. Dans ce modèle, une structure plus précise d'un composant adaptatif est présentée et surtout il est question de sélection d'algorithmes optimaux. Le but de l'adaptation est alors de trouver « la meilleure solution » pour la situation qui se présente au logiciel.

2.3.3 Paramétrisation par interfaces ajustables

Chang et Karamcheti [6] proposent une approche par interface ajustable qui vise à rendre disponibles des paramètres de contrôle, similaires à des boutons de réglage, pour modifier le fonctionnement interne d'un composant. L'ensemble des valeurs des paramètres d'un composant est considéré comme une configuration de ce composant. En soumettant les composants à différentes situations liés aux ressources de l'environnement d'exécution, le système d'adaptation peut lier leur effet à des métriques de qualité de service (QdS) et construire un modèle qui permettra de satisfaire des contraintes. Plusieurs contraintes peuvent être définies et une fonction d'objectif est utilisée pour quantifier leur importance relative. Il s'agit au final d'un problème d'optimisation de cette fonction qui est résolu par son évaluation sous diverses configurations à l'initialisation du logiciel ou avant (lors d'une exécution antérieure). La contribution principale de cet article est l'idée de paramétrer des composants et de contrôler cette paramétrisation par l'identification de configurations appropriées de façon automatique. Plutôt que de remplacer un composant ou un algorithme, on cherche plutôt à injecter à un composant un état de paramétrisation différent. Une attention particulière a été portée dans cette recherche à l'élaboration d'une méthode transparente, c'est à dire qui ne nécessite pas de modification aux interfaces exposées. Ce souci pour la transparence permet notamment l'ajout plus facile de ce type de fonctionnalités à des logiciels hérités (« *legacy software* ») et à des librairies (limitant les modifications aux applications qui en sont dépendantes).

2.3.4 Modèle d'architecture par graphe

L'idée d'un composant adaptatif s'est popularisée dans les années 2000 et plusieurs chercheurs ont proposé les responsabilités potentielles d'un tel composant. Des façons de faire la gestion des interactions et réorganisations entre ces composants ont également été pro-

posées. Souvent, une certaine modélisation de l'architecture d'un logiciel est utilisée pour faciliter l'analyse de cette dernière et identifier des problèmes. Par exemple, Georgiadis *et al.* [14] utilise l'approche proposée par le modèle Darwin qui consiste à voir l'architecture d'un système sous forme de graphe orienté avec ses composants comme sommets et les liaisons entre les composants offrant un service et ceux l'utilisant comme arcs. Chaque composant est branché à d'autres via des ports, sortants ou entrant en fonction de la nature de la dépendance au service. Il s'agit dans ce cas de sélectionner les meilleurs composants dont les ports sont compatibles pour construire l'architecture. Dans ce modèle, l'entité qui a la responsabilité d'effectuer l'adaptation de l'architecture est souvent découplée des composants. Cette entité est appelé un « gestionnaire de composants » Georgiadis *et al.* [14] et met en place le contrôle externe, en opposition au contrôle interne qui attribue toutes les responsabilités d'adaptation au composant adaptatif [43]. Un avantage de ce type de contrôle est que la mise en place de l'adaptation ne nécessite pas de modification aux composants d'un système. De plus, une vue de la configuration du système est construite et permet au gestionnaire de composant d'acquérir des connaissances plus globales du système et de son architecture. Cette vue est maintenue à jour grâce à un mécanisme de notifications lorsque des changements d'architecture sont effectués. Le système est gardé dans un état connu lorsque des opérations sont en cours et un verrou doit être acquis pour effectuer une adaptation (empêchant des opérations pendant l'adaptation). Tout comme pour [6], la réorganisation automatique est un problème d'optimisation basé sur des contraintes.

2.3.5 Modèle MAPE-K

La compagnie IBM a introduit en 2003 puis développé dans les années suivantes un modèle complet visant l'implémentation d'architectures adaptatives : MAPE-K [20] (voir figure 2.2). Les lettres du modèle forment un acronyme pour les mots *Monitor, Analyze, Plan, Execute* et *Knowledge* qui sont différentes préoccupations d'un système adaptatif et dont les quatre premiers forment une boucle de rétroaction. Bien que ce modèle est issu du milieu commercial, il eut beaucoup d'influence sur le domaine de la recherche et beaucoup de modèles s'en sont inspirés. Notamment, un article très influent de Hinchey et Sterritt [17] cite le modèle comme une implémentation des principes de base d'un système autonome (conscient de lui-même et de son environnement, capable de détecter des changements dans ces derniers et de changer son comportement en conséquence). Dans le modèle MAPE-K, un gestionnaire autonome (*autonomic manager*) est une entité dont la responsabilité est de coordonner les activités des différents modules de la boucle de rétroaction. Un moniteur

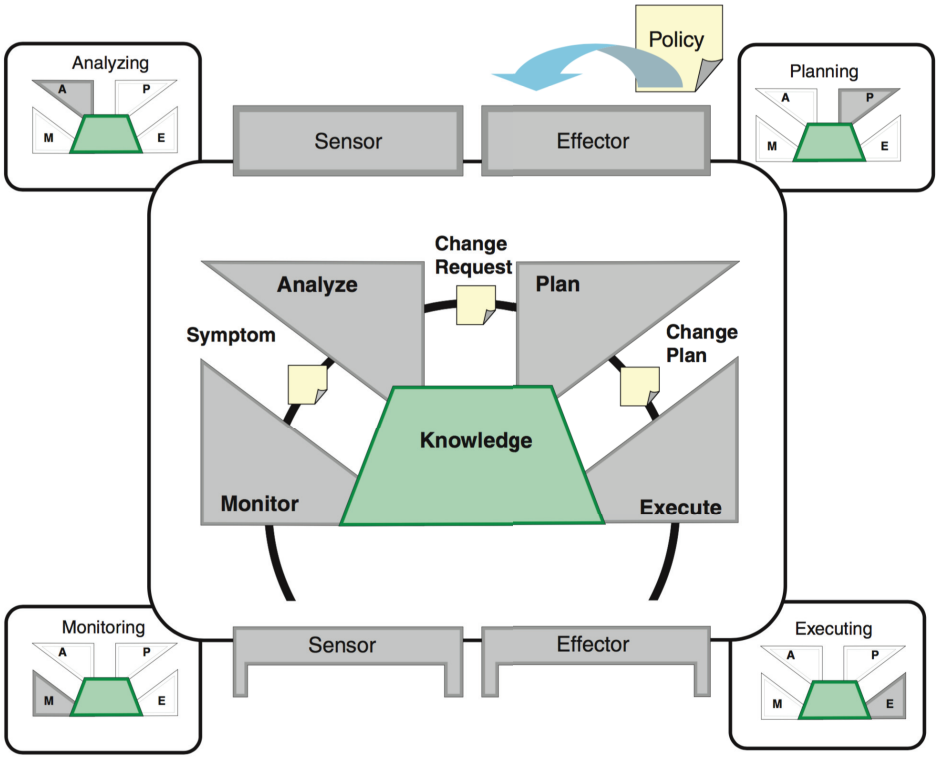

Figure 2.2 Modèle MAPE-K d'IBM (tiré de [20])

récupère des informations du système et détecte certains symptômes en vue d'être analysé pour déterminer si des actions doivent être entreprises. Si des actions sont nécessaires, un plan est formé pour mettre en place une adaptation du système qui est finalement exécuté à un moment opportun. La boucle est complétée alors que des changements sont mis en place dans le système et que les moniteurs peuvent obtenir de nouvelles données. Toutes les informations utilisées dans l'exécution des différentes fonctions de la boucle de rétroaction forment ensemble une base de connaissances qui est partagée à l'intérieur d'un même gestionnaire autonome. Une contribution importante de ce modèle est la séparation claire des boucles de rétroaction qui, dans les précédentes recherches, n'ont été qu'implicites. Également, la réalisation que plusieurs de ces boucles peuvent être définies de façon hiérarchique permet de plus facilement spécifier des comportements adaptatifs complexes. Par le fait même, chaque composant peut ainsi être géré par un gestionnaire autonome distinct. Tout comme [14], la gestion externe est adoptée et des points de surveillance et d'action dans le sous-système géré sont définis explicitement sous forme de capteurs et d'effecteurs. Dans ce cas, les liens entre le gestionnaire et son sous-système sont donc définis à la conception. Additionnellement, plutôt que de définir un format abstrait d'un composant adaptatif, il est possible de superposer à des composants existants des mécanismes d'adaptation sans modifier le système initial (l'adaptation est la préoccupation exclusive des gestionnaires

autonomes). Finalement, l'applicabilité du modèle à un vaste domaine de systèmes est accrue par sa flexibilité au niveau de la spécification des stratégies utilisées à chaque étape de la boucle de rétroaction. La formalisation de ces étapes a été une contribution massive de ce modèle.

2.3.6 Amélioration des modèles dans les prototypes et *frameworks*

Plusieurs recherches par prototype ont visé le développement de librairies qui appliquent différentes variations des modèles présentés dans les années précédentes. Plutôt que de proposer de nouvelles méthodes ou modèles, ces recherches avaient pour but l'optimisation des procédés d'adaptation et le développement d'une architecture logicielle concrète utilisable à grande échelle pour minimiser les risques rattachés à l'inclusion de l'adaptation dans les logiciels. Les avantages au niveau de la QdS pour des systèmes complexes comme les systèmes distribués ont certainement influencé les recherches en étant les principaux sujets de cas d'étude. Les librairies développées ont largement été influencées par les besoins spécifiques de ces applications.

Plusieurs solutions ont été développées sous forme de *frameworks*, soit des éléments de conception logiciels incluant souvent une librairie, un langage intermédiaire et différents outils supplémentaires. Ces *frameworks* proposent des améliorations qui sont importantes par rapport aux principes originaux qui doivent être pris en compte. Quelques exemples de ceux-ci sont *Rainbow framework* proposé par Garlan *et al.* [13] et *Accord* proposé par Liu et Parashar [25]. D'autres *frameworks* avec un rôle connexe ont également été proposés. Quelques exemples sont *Ceylon* proposé par Maurel *et al.* [31] qui permet la conception de gestionnaires autonomes plutôt que d'applications adaptatives ainsi que le *framework* conceptuel proposé par Malek *et al.* [29] qui permet d'effectuer la conception de système à architecture distribuée en vue de satisfaire à des caractéristiques précises. Bien que chacun d'entre eux apporte des améliorations, il est plus pertinent d'analyser des travaux de synthèse de ces solutions afin de déceler les éléments récurrents et qui sont généralisables pour une vaste variété d'applications adaptatives.

2.4 Patrons de conception

Les patrons de conception constituent un moyen de partager des solutions à des problèmes récurrents en informatique par le biais de schémas et d'explications dans une forme standardisée. La forme la plus connue est celle utilisée par le *Gang of Four* dans leur livre *Design Patterns : Elements of Reusable Object-oriented Software* [12]. Dans le cas du pré-

sent projet de recherche, le problème récurrent est l'ajout graduel de l'adaptation dans une application. En général, les travaux ayant comme sujet les patrons de conception dans le domaine de l'adaptation dans les logiciels sont variés et touchent divers domaines d'application.

Plusieurs modèles visant la spécification d'une structure standard d'architectures de composants adaptatifs ont été présentés à la section 2.3 et pourraient être réinterprétés sous forme de patrons. Ces modèles manquent cependant de précision au niveau de l'implémentation et, vu leur âge, ne bénéficient pas de l'expérience acquise plus récemment pour le développement de logiciels adaptatifs, particulièrement poussé par le développement de *frameworks* (voir section 2.3.6).

La présente section a pour but de présenter les patrons de conception tirés de cette expérience et des défis supplémentaires qu'apportent les systèmes adaptatifs complexes. En lien avec le présent projet de recherche, il s'agit d'analyser les éléments communs et suffisamment généraux pour être essentiels à la mise en place de l'adaptation. Une description des concepts essentiels retenus pour la proposition des patrons a été réalisée dans le premier article qui est reproduit à la section 4.2.

2.4.1 Composant virtuel

Corsaro *et al.* [10] proposent une solution à un problème récurrent des composants réutilisables : de quelle façon peut-on faire la conception d'un composant qui est réutilisable et peut cibler plusieurs systèmes très différents sans complexifier accidentellement l'interface qu'il expose ? La proposition est sous forme d'un patron de conception qui est appelé *composant virtuel* (illustré à la figure 2.3). Il s'agit essentiellement d'une concrétisation de la technique décrite dans [7] : un composant logique (CAM) agi en mandataire ("*proxy*") et redirige les appels vers un composant concret (AAM) qui est le mieux approprié à la situation dans laquelle l'appel est exécuté (voir section 2.3.2). La contribution supplémentaire de cette recherche est la proposition d'associer des stratégies de chargement et de déchargement à chaque composant concret et d'attribuer la responsabilité de l'exécution de ces stratégies à un module fabrique (du patron de conception "*factory*"). Cette plus grande séparation des préoccupations et le fait de fournir avec le composant tout le nécessaire pour gérer son cycle de vie permettent de régler efficacement le problème énoncé plus tôt. Il est également important de remarquer que l'interface initiale est gardée intacte, suivant donc les traces de [6] sur la transparence au niveau de l'interface exposée. Différentes stratégies de chargement et de déchargement sont également explorées, principalement dans le but d'offrir un meilleur contrôle sur les ressources utilisées par une application.

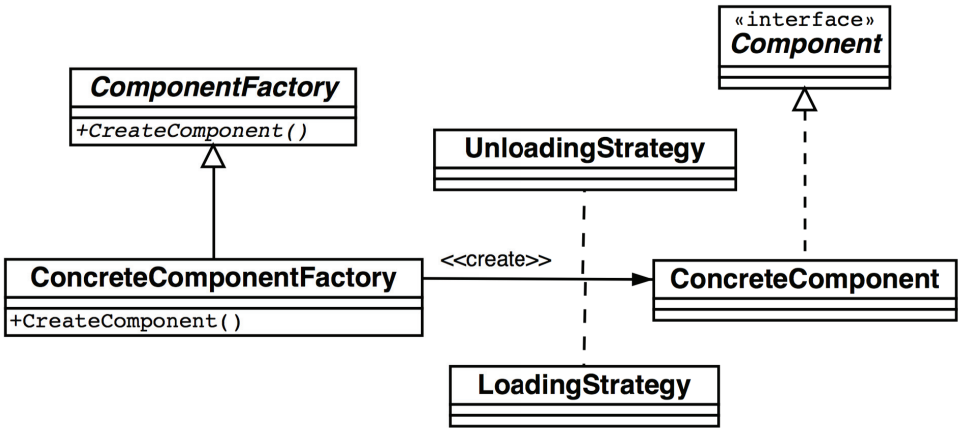

Figure 2.3 Patron « composant virtuel » (tiré de [10])

2.4.2 Patrons liés aux préoccupations de l'adaptation

Les patrons de conception qui sont le plus dans l'esprit de ceux proposés à l'origine par le *Gang of Four* sont ceux de Ramirez [39]. Ces derniers sont regroupés par préoccupation et ne sont pas liés directement à un domaine d'application précis. Les patrons proposés sont issus d'une recherche intensive basée sur des projets libres, commerciaux et de recherche. L'approche décrite dans les travaux de Ramirez consiste à relever des solutions à des problèmes récurrents des logiciels adaptatifs. Ces solutions spécifiques aux domaines respectifs des projets analysés sont généralisées et affinées afin qu'elles puissent être utilisées dans n'importe quel domaine d'application. Au total, 12 patrons sont proposés et présentés dans un format standardisé qui couvre des caractéristiques comme le contexte d'utilisation, la structure du patron, les éléments le composant et les contraintes pour son implémentation. Les patrons et leurs relations sont présentés à la figure 2.4. Cette figure représente une organisation des patrons qui permet la création d'un logiciel adaptatif selon différentes stratégies. Un développeur peut ainsi choisir depuis des requis d'adaptation les patrons les plus adéquats pour chacune des préoccupations.

Bien que les patrons proposés par Ramirez sont applicables à divers domaines, plusieurs d'entre eux restent liés au type d'application développé. Par exemple, le patron *server reconfiguration* reste spécifique aux systèmes client-serveur. Aussi, plusieurs artéfacts sont présents dans plus d'un patron. Par exemple, les éléments *component*, *inference engine* et *rule* font partie de trois patrons de type *decision-making*. Ces spécificités indiquent que la structure qu'ils induisent au logiciel cible peut être très différente en fonction du domaine tout en partageant, malgré tout, plusieurs éléments clés. Bien qu'un effort de généralisation a été fait, aucune structure de base ne peut être identifiée. Cela implique que l'implémentation de chacun des patrons peut amener à une architecture largement

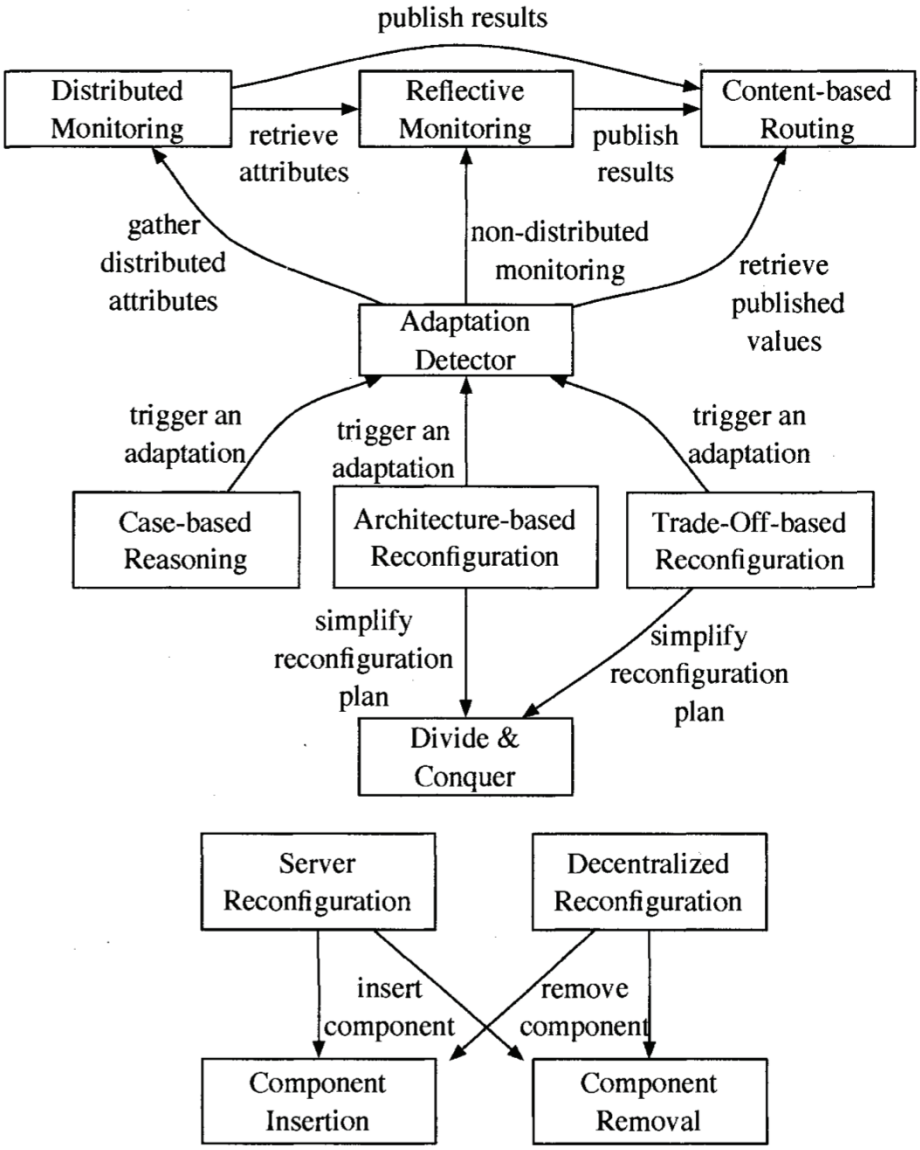

Figure 2.4 Relations entre les patrons proposés par Ramirez (Tiré de [39])

différente et rendre difficile la comparaison des différentes solutions pour chacune des préoccupations. Ramirez offre tout de même une synthèse des solutions qui reste la plus complète à ce jour au niveau des patrons de conception.

2.4.3 Patrons pour structures spécifiques

D'autres projets de recherche ont proposé des patrons de conception pour différents problèmes. Cependant, ils sont généralement plus liés à un domaine d'application ou ciblent un contexte plus spécifique que les logiciels adaptatifs. Quelques travaux de cette nature sont présentés dans la présente section.

D'abord, Mannava et Ramesh [30] ont repris l'essentiel des patrons proposés par Ramirez et ajouté trois autres : *row data gateway*, *thread per connection* et *event notifier*. Typiquement, les deux premiers patrons sont utilisés pour des systèmes client-serveur tandis que la notification par évènement est utilisée accessoirement pour découpler les modules contrôleurs des composants cibles à la façon du patron *content-based routing* de Ramirez.

Ensuite, Menasce *et al.* [33] ont proposé des patrons architecturaux pour la QdS de systèmes adaptatifs. L'idée est d'améliorer la QdS en maximisant des *fonctions utilitaires* par le biais d'adaptations du système. Ces fonctions ont comme arguments des propriétés du système qui sont mesurées à l'exécution par un artéfact nommé *monitor*. Le but d'un tel système adaptatif consiste donc à planifier des reconfigurations de la connexion de composants pour atteindre des valeurs cibles des fonctions utilitaires. Les patrons présentés sont donc en réalité des structures pour mettre en place des stratégies de QdS comme la répartition de charge (*load balancing*), ou la division pour régner (*divide and conquer*). Au final, la structure mettant en place l'adaptation est dans tous les cas une architecture par composants dont l'organisation est choisie parmi un ensemble de configurations fonctionnellement équivalentes, chacune liée à une valeur de qualité induite au système. En soi, cette structure est une solution à l'ajout de comportement adaptatif dans un système et les patrons l'utilisent pour proposer des solutions liées à la QdS. Le problème est que les patrons ne peuvent s'appliquer aux logiciels dont la structure est trop différente.

Finalement, Weyns *et al.* [47] ont proposé un ensemble de patrons de contrôle décentralisé de l'adaptation : *coordinated control*, *information sharing* et *regional planning*. Ces patrons supposent une structure dont les préoccupations ont été séparées suivant le modèle MAPE-K. L'application de ce modèle est cependant interprétée dans une classification des patrons réalisée par Berkane *et al.* [2]. Ces derniers classifient huit patrons de conception en fonction de leur responsabilité en lien avec les étapes du modèle MAPE-K et orthogona-

lement en niveau d'implémentation. La majorité des patrons sont tirés de [39, 40] et sont classifiés au niveau logique, c'est-à-dire qu'ils sont plus abstraits que les patrons de conception classiques (tirés de [12]), eux-mêmes classifiés à la couche technique. Cependant, ils sont moins abstraits que les étapes du MAPE-K qui sont classifiées au niveau fonctionnel. Il n'est cependant pas question d'analyser la structure induite par l'utilisation des patrons de conception et donc de déterminer si cette structure implémente correctement et complètement le modèle.

Les limitations et spécifications des patrons de conception mènent donc à l'idée de proposer une structure la plus standard possible pouvant servir de base et où les différents mécanismes de l'adaptation comme ceux présentés précédemment puissent être efficacement ajoutés, modifiés et comparés. En d'autres mots, il devrait être facile d'assembler des composants présentant des propriétés d'adaptation en un système complet et de modifier cette adaptation en affectant le moins possible le reste du système. Cette séparation des préoccupations est centrale à la qualité logicielle. Aussi, le niveau de granularité des patrons de conception est plus propice celui des modèles présentés à la section 2.3 pour aider les développeur dans l'implémentation de l'adaptation.

2.5 Vérification et de validation de logiciels adaptatifs

Le développement de techniques de vérification et validation (V&V) des logiciels adaptatifs est un défi de taille. Il a été indiqué par Lemos *et al.* [24] qu'un des défis principaux est la recherche de moyens pour effectuer la vérification d'un logiciel adaptatif à l'exécution dans un temps tout en contrôlant la complexité qui en découle inévitablement. Des techniques comme la vérification de modèles et la vérification quantitative à l'exécution sont mentionnées par Calinescu *et al.* [5]. Ils expliquent également comment la sélection de services à l'exécution peut être contrôlée par un outil de vérification quantitatif pour s'assurer que les spécifications et les requis sont continuellement satisfaits.

Les métriques de qualité sont un moyen d'effectuer la vérification dans les logiciels par rapport à une préoccupation. Une métrique produit un indicateur qui peut être interprété par rapport à un domaine de valeurs. Il est ainsi possible d'indiquer des valeurs cibles et détecter si l'implémentation présente des problématiques et pourrait être améliorée. Bien que typiquement utilisées à la conception, elles peuvent également être utilisées à l'exécution. Dans le cas d'un logiciel adaptatif, les valeurs qu'elles fournissent pourraient par exemple être utilisées comme facteurs décisionnels dans une stratégie d'adaptation.

La présente section introduit les différentes approches actuellement utilisées pour la V&V des logiciels adaptatifs. Au niveau des métriques, la majeure partie de celles proposées sert à comparer différents logiciels adaptatifs et est généralement qualitative plutôt que quantitative. Plus de détails sur ces métriques sont donnés à la section 6.2.2.

2.5.1 Phase de conception

Dans les logiciels adaptatifs, plusieurs techniques de V&V ont été proposées, soit pour la phase de conception du système ou à l'exécution. À la conception, une approche populaire est la vérification d'un modèle du système. Par exemple, Liu *et al.* [26] propose « une approche hybride pour la détection efficace de fautes d'adaptation » pour les applications adaptatives. Leur approche suppose une stratégie d'adaptation contrôlée par des règles d'adaptation. Pour diminuer la complexité de l'exploration de tous les états d'adaptation, ils utilisent une approche probabiliste : une valeur de probabilité est donnée suite à une exécution du système à chaque paire d'*atomes propositionnels* (chaque élément d'un contexte qu'ils modélisent comme une valeur booléenne). La valeur de probabilité est un indicateur de confiance envers une contrainte formée des deux atomes propositionnels. Par exemple, si deux atomes a et b ont toujours le même état lors de l'exécution, le taux de confiance est très haut. Ainsi, il est possible de spécifier une contrainte où tout état du système incluant cette paire à des valeurs différentes n'est jamais rencontré en pratique.

L'idée derrière cette technique ainsi qu'une multitude d'autres est de minimiser le plus possible l'espace d'états devant être vérifié. En d'autres mots, les zones de l'espace d'état du système où les contraintes sont violées n'ont pas besoin d'être vérifiées ce qui accélère la vérification.

2.5.2 Phase d'exécution

Cette phase est celle qui concerne le plus les logiciels adaptatifs, car le dynamisme qu'induit l'adaptation rend les méthodes de V&V traditionnelles insuffisantes. Il est nécessaire de différencier les éléments qui sont la cible de la V&V. En effet, un développeur pourrait vouloir vérifier l'atteinte de buts d'adaptation, typiquement relié à la QdS, mais aussi vouloir vérifier si l'implémentation du logiciel adaptatif garanti que tous les états atteints lors de l'exécution seront supportés. Beaucoup de travaux proposent des métriques pour les propriétés de performance et de QdS [22, 38].

Pour certains, l'adaptation est en soi un moyen de mettre en place une assurance qualité dans les systèmes. Par exemple, Rushby [42] propose que la certification de systèmes

puisse être atteinte par l'utilisation de mécanismes effectuant des vérifications à l'exécution et détectant des anomalies. La définition d'une anomalie est cruciale à la validation du système. En fait, une anomalie peut être une déviation du comportement par rapport à un modèle généré lors de tests [42], mais comme à la phase de conception une anomalie pourrait être un état d'adaptation qui viole des spécifications ou un état pour lequel le comportement du système n'est pas défini.

L'analyse de la zone de viabilité (l'ensemble des états du système qui permettent l'atteinte des requis et spécifications) est une forme de vérification de modèle similaire à celle utilisée à la conception puisque cette zone doit être définie depuis un modèle représentatif du système à l'exécution. Cependant, contrairement à la phase de conception, la zone de viabilité peut changer en fonction de l'état d'adaptation du système [24, 44], rendant possible l'atteinte d'états hors de cette zone. Des techniques sont donc nécessaires pour évaluer la zone de viabilité pour chacun des états du système. Surtout, il est nécessaire de pouvoir indiquer au système si ce dernier ne se trouve pas dans la zone de viabilité afin que des actions adaptatives puissent permettre à ce dernier de le redevenir.

2.6 Enquêtes sur le domaine

Les enquêtes scientifiques sont utiles pour obtenir un aperçu d'un domaine ainsi que les défis auxquels ce dernier fait face à un moment. Surtout, il est important de considérer les défis identifiés pour centrer les recherches effectuées dans le domaine. Pour le présent projet de recherche, les enquêtes ont servi à mieux préciser les éléments de recherche. Dans cette section, deux enquêtes principales sont présentées et les aspects qui touchent le plus le présent projet de recherche sont résumés. Elles sont également utilisées pour expliquer la façon dont les contributions sont utiles pour l'avancement du domaine.

D'abord, l'enquête réalisée en 2013 qui constitue la plus récente est présentée à la section 2.6.1. Par la suite, une enquête réalisée en 2009 est présentée à la section 2.6.2.

2.6.1 Lemos *et al.*

Dans l'enquête de Lemos *et al.* [24], quatre sujets principaux sont relevés comme étant d'intérêt pour les recherches futures : « conception des solutions adaptatives, procédés d'ingénierie des logiciels adaptatifs, décentralisation des boucles de contrôle et V&V pratique à l'exécution des logiciels adaptatifs » Lemos *et al.* [24]. Deux de ces aspects sont en

liens étroits avec le présent projet de recherche : procédé d'ingénierie et V&V. Ces derniers sont résumés depuis [24] :

Procédé d'ingénierie

Les logiciels adaptatifs sont différents des logiciels traditionnels puisqu'ils opèrent dans « un monde hautement dynamique et doivent ajuster automatiquement leur comportement en réponse aux changements dans leur environnement, leurs buts ou dans le système lui-même. » Lemos *et al.* [24] Dans le cycle de vie traditionnel d'un logiciel, les développeurs ont le rôle de transformer un logiciel pour atteindre les requis d'un client. Une fois le logiciel déployé, le comportement du logiciel est fixé et des changements dans les requis impliquent inévitablement le travail des développeurs. Dans le cas des logiciels adaptatifs, le rôle des développeurs est modifié, passant « d'opérationnel à stratégique » Lemos *et al.* [24]. L'impact de cette modification est que les phases de développement traditionnelles ne sont plus aussi représentatives du travail à entreprendre. En effet, comme les requis sont analysés à l'exécution, le défi des développeurs est d'effectuer l'ingénierie des éléments mettant en place l'adaptation et les mécanismes pour effectuer le contrôle du système. Pour ce faire, la modélisation du système et de son environnement doit faire partie du cycle de vie du logiciel. Une partie de cette modélisation peut se faire en partie à la conception et ce modèle peut être modifié de façon autonome pour reproduire le système physique tel qu'il est dans son environnement à l'exécution.

Le lien principal avec la technique d'ajout de comportement adaptatif proposée dans le présent projet de recherche est au niveau de la planification. En effet, l'évolution d'un logiciel qui s'adapte à son environnement doit se faire conjointement à la modélisation de ce même environnement. Plus particulièrement, la phase d'implémentation d'une fonctionnalité peut inclure ou non un comportement adaptatif. Il faut, cependant, bien différencier les requis liés à une fonctionnalité de ceux liés à l'adaptation comme l'autoguérison. Avec l'utilisation des patrons de conception, les composants sont d'abord développés pour une fonctionnalité spécifique *et* un espace de contextes défini. Un composant peut donc être développé avec un comportement adaptatif contraint à l'intérieur de cet espace, mais l'utilisation de mécanismes comme la substitution de composants est gérée dans une phase ultérieure. Cette approche de développement permet donc un développement en intégration continue où l'espace de contextes que le système est attendu de supporter est couvert graduellement. Le modèle développé est également fait en deux phases qui peuvent être intégrées séparément dans les phases de développement : à la conception initiale (modélisation des paramètres) et à la spécialisation pour une plateforme (modélisation des moniteurs).

Vérification et Validation

Un problème important dans le domaine de l'adaptation est le développement de la confiance des utilisateurs envers les décisions prises de façon autonome par le système. La certification par des techniques de V&V est alors utilisée comme façon de prévoir la validité du comportement d'un logiciel adaptatif et d'améliorer la confiance des utilisateurs. De telles techniques sont difficiles à élaborer. La tâche est encore plus ardue lorsqu'un procédé traditionnel impose une assurance de la satisfaction des requis de façon explicite à la conception. L'inclusion des techniques de V&V à l'exécution comme information utilisée dans les décisions du système est une piste envisagée pour faire en sorte, par exemple, que l'atteinte d'états jugés problématiques soient prévenue de façon automatique. L'ensemble des états du système qui permettent à ce dernier d'atteindre ses buts (ses « requis et propriétés désirées » Lemos *et al.* [24]) est appelée une « zone de viabilité » Aubin *et al.* [1]. Au final, il est donc question de vérifier si le système est dans cette zone de viabilité. Si ce n'est pas le cas, il serait alors désirable de reconfigurer le système pour tenter d'atteindre un état dans la zone de viabilité. Tout comme d'autres phases de développement, le V&V passerait ainsi partiellement de la conception à l'exécution.

Les métriques qui sont proposées dans le présent projet de recherche sont une tentative d'offrir un outil de vérification pour la structure de base livrée par l'utilisation des patrons de conception proposés. La différence majeure avec les défis et solutions potentielles relevés dans le paragraphe précédent concerne le niveau d'assurance que nous visons. Plutôt que de vérifier l'atteinte de propriétés désirées au niveau du système, l'hypothèse proposée est de réaliser cette vérification au niveau des composants. Par la composition de différents composants offrant un même service, l'argument utilisé dans le présent projet est que les zones de viabilité de chacun des composants peuvent être combinées. Ainsi, la zone de viabilité d'un composant agglomérant ces différents « candidats » est représentée par l'union des contextes de ces derniers. Les métriques proposées servent à calculer la proportion des états qui sont couverts par un composant de sorte que les situations où un état indéfini existe puissent être détectées. Les métriques proposées se veulent donc des outils pour assurer qu'un système respecte ses spécifications en termes de zone de viabilité telle que modélisée. De plus, chacune des phases de conception et d'exécution utilise une modélisation de la zone de viabilité différente. Le V&V n'est donc pas entièrement remis à l'exécution : il est appliqué aux différentes phases.

2.6.2 Cheng *et al.*

L'enquête de Lemos *et al.* [24] est la suite d'une enquête réalisée par Cheng *et al.* [8] quatre ans plus tôt. Cette première enquête contenait une analyse de quatre grands aspects du domaine : les dimensions de modélisation, les requis, l'ingénierie et l'assurance des logiciels adaptatifs. L'ingénierie et l'assurance sont les deux aspects qui rejoignent ceux discutés précédemment dans la section 2.6.1. Les défis qui y sont reliés sont tout de même d'actualité. En lien avec les aspects présentés à la section 2.6.1, des défis qui avaient été relevés par cette enquête sont présentés.

Ingénierie

Ré-ingénierie : « Explorer les techniques pour l'évolution des systèmes existants et l'injection de l'adaptation dans de tels systèmes » Cheng *et al.* [8].
Ce défi est relevé par les patrons qui offrent une façon d'ajouter graduellement des comportements adaptatifs tout au long du cycle de vie d'un logiciel adaptatif.

Support *middleware* : Le développement d'une implémentation incluant « des interfaces standardisées pour les fonctionnalités reliées à l'adaptation » Cheng *et al.* [8].
Ce défi est relevé par le développement de la librairie de référence AdaptivePy où les artéfacts et leur interface sont standardisés dans les patrons de conception.

Interaction personne-machine : Inclure les utilisateurs dans les boucles de rétroaction pour « assurer la confiance » Cheng *et al.* [8] envers le système.
Ce défi est abordé grâce à l'application de l'adaptation dans le contexte des interfaces graphiques. Ceci est réalisé concrètement par le biais d'une étude de cas. Cette dernière prévoit l'utilisation des interactions de l'utilisateur comme données contrôlant l'adaptation. Cependant, la boucle de rétroaction n'est pas modifiable par l'utilisateur directement et le prototype de l'étude de cas simule les interactions précédentes d'utilisateur pour faciliter l'observation de l'adaptation.

Vérification et Vérification

Environnements contrôlés par modèles qui sont spécifiques à l'adaptation : Le développement par modèles est une piste pour permettre « l'application de méthodes de V&V pendant le processus de développement et le support de l'adaptation à l'exécution. » Cheng *et al.* [8] L'idée est d'utiliser ce modèle pour prévoir les impacts sur le système qu'aura l'adaptation.
Ce défi est en partie relevé par la proposition du modèle par espace d'états pour

spécifier les domaines de viabilité et leur utilisation dans les composants adaptatifs et dans les artéfacts du patron moniteur. Particulièrement, ce patron permet la surveillance des changements dans le système lors de l'exécution. Les métriques peuvent également être utilisées à l'exécution pour détecter des états non définis et donc sont une forme de vérification du modèle à l'exécution.

Assurance à l'exécution agile : « Le requis clé pour la vérification à l'exécution est l'existence de solutions algorithmiques agiles efficaces qui ne requièrent pas une haute complexité temps/espace. » Cheng *et al.* [8]
Ce défi est abordé par la méthode de calcul liée aux métriques développées. En diminuant le domaine de calcul à deux niveaux de composition et en utilisant une structure abstraite d'espace d'états, la vérification d'architectures complexes est possible de façon pratique. Cependant, l'implémentation au niveau du calcul d'états récurrents n'est pas optimale. Or, plus de travail est requis pour développer une solution encore plus performante.

CHAPITRE 3

Développement des patrons de conception

Un premier objectif secondaire identifié à la section 1.3 vise la proposition de patrons de conception afin de mettre en place des comportements adaptatifs de façon graduelle. En vue de la rédaction d'un premier article à ce sujet, plusieurs étapes ont été nécessaires pour analyser et élaborer les patrons de sorte que ceux-ci soient le plus universels possible. De plus, le développement de la librairie de référence AdaptivePy fut partie intégrante du travail de test et de validation des solutions. Ce chapitre vise à introduire tous les éléments de développement qui entourent la réalisation des patrons de conception qui sont présentés dans l'article 1, reproduit intégralement au chapitre 4.

3.1 Méthodologie

Le présent projet de recherche a nécessité la recherche et la synthèse de concepts généralisés dans le domaine de l'adaptation des logiciels ainsi que la recherche de patrons de conception utilisant ces concepts. Cette section présente la méthodologie de recherche qui a mené à la rédaction du premier article.

3.1.1 Approche générale

L'approche générale appliquée à ce projet de recherche est basée sur une démarche itérative, autant au niveau de l'analyse que des développements. Cette démarche permet de mettre en pratique les concepts identifiés séparément et d'analyser les limites de leur implémentation. Au fur et à mesure que les concepts sont identifiés dans la littérature, ils sont utilisés dans un contexte réel comme solution partielle et les avantages relevés peuvent alors guider la solution générale qui formera le patron. Un schéma résumant les étapes d'une itération est donné à la figure 3.1.

Pendant une période donnée d'environ deux semaines, un prototype était développé afin de mettre en place une préoccupation de l'adaptation ou pour en améliorer une déjà mise en place précédemment. Les artéfacts qui sont liés à cette préoccupation peuvent ainsi être ajoutés ou améliorés pour mitiger une lacune identifiée précédemment. Un artéfact fait ici référence à un élément de solution modélisé qui compose un patron. Par exemple, un élément de solution permettant de traduire en langage contextuel de plus haut niveau le

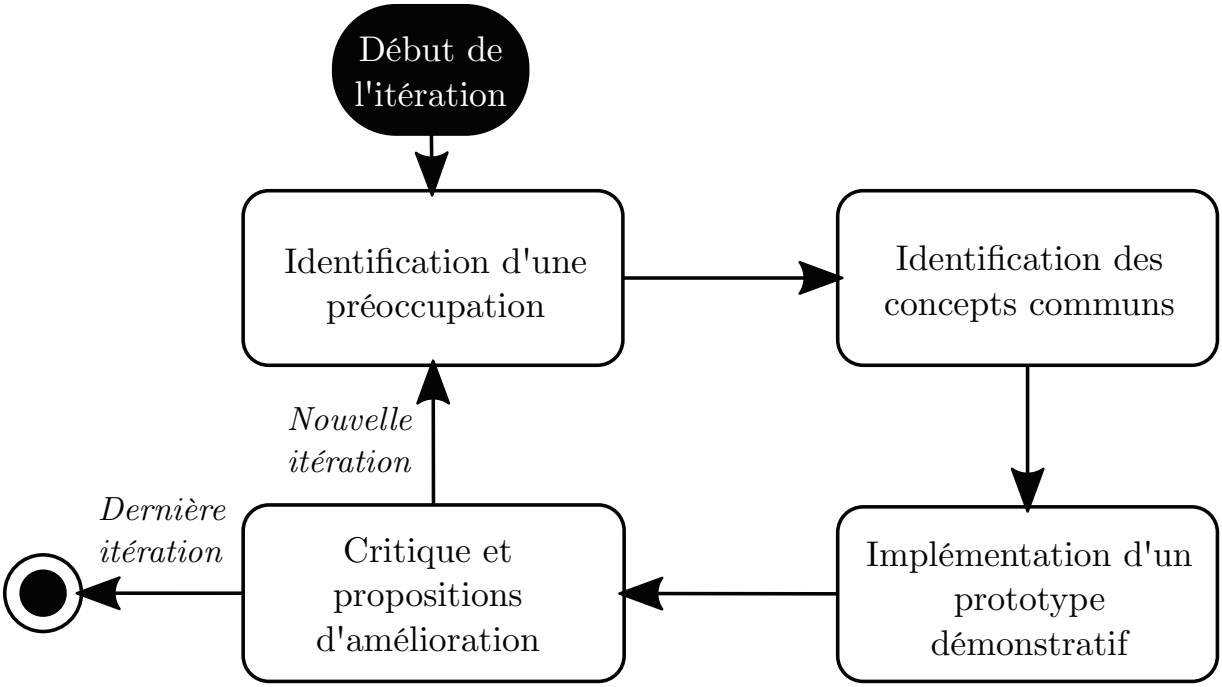

Figure 3.1 Démarche itérative pour le développement des patrons de conception

contexte actuel depuis des données brutes de capteurs serait considéré comme un artéfact. Pour chaque fin de période, une mise à jour de la synthèse des concepts était produite et utilisée comme base pour proposer une version des patrons de conception. L'essentiel des concepts était considéré comme couvert lorsqu'il était possible de produire une application complète sans nécessairement spécifier un mécanisme d'adaptation précis. Une implémentation de référence des artéfacts tirés des concepts a été faite dans la librairie AdaptivePy, en partie en se basant sur les prototypes des itérations précédentes. Pour mieux évaluer les patrons finaux, une application adaptative graphique a été développée avec et sans la librairie. Les sections suivantes précisent un peu plus chaque point énoncé précédemment.

3.1.2 Acquisition et synthèse des concepts essentiels

Pour proposer des patrons de conception, il est nécessaire d'identifier des concepts fondamentaux qui sont partagés par beaucoup de systèmes adaptatifs. Ces concepts doivent être suffisamment généraux pour être inclus sous forme d'artéfacts logiciels qui seront utilisés pour implémenter différents comportements adaptatifs. Ainsi, des types de comportements adaptatifs doivent être également identifiés et liés aux concepts pour s'assurer que leur implémentation est possible avec les artéfacts en question.

Comme l'approche générale est itérative, l'acquisition et l'analyse de l'état de l'art se sont faites par préoccupation et en même temps que les développements. La première itération a été dédiée à la lecture des enquêtes dans le domaine et des travaux principaux sur la question des patrons de conception, par exemple le mémoire de Ramirez [39]. Par la suite, une préoccupation cible a été identifiée avant chaque itération afin de concentrer les recherches pendant ces périodes. Les types de travaux analysés inclus les propositions de patrons de conception, de *frameworks*, de modèles, de concepts d'adaptation ainsi que de prototypes présentant des mécanismes d'adaptation.

Pour le premier article uniquement, près de 80 documents ont été analysés et 24 d'entre eux ont été retenus comme apportant des concepts généralisables et originaux. La synthèse des concepts est présentée dans l'article à la section 4.2. Trois préoccupations sont retenues comme générales : la surveillance des données d'adaptation, les méthodes d'adaptation dans les composants et les stratégies d'adaptation. Il est à noter que les patrons visent principalement les applications graphiques, ainsi il est question plus spécifiquement de *composants* pour les méthodes d'adaptation.

Aussi, ce que la littérature identifie généralement comme *mécanismes* d'adaptation est identifié comme *stratégies* d'adaptation. Cette distinction a été faite pour mettre en évidence la différence entre mécanisme et méthode d'adaptation : une méthode représente la maniabilité de la structure tandis que le mécanisme est la stratégie utilisée par le système pour atteindre ses buts d'adaptation. En effet, une stratégie tire parti des méthodes d'adaptation disponibles pour atteindre les buts d'adaptation.

Pour déterminer si un concept est généralisable, il devait se souscrire aux requis suivants :

- Être neutre par rapport aux mécanismes/stratégies d'adaptation
- Être neutre par rapport au domaine d'application
- Couvrir une préoccupation de l'adaptation partagée par la majorité des modèles comme MAPE-K et des *frameworks*

3.1.3 Validation par prototypes

Afin d'assurer un lien cohérent entre les concepts identifiés et leur application, des prototypes mettant à exécution des préoccupations de l'adaptation ont été produits pour chacune des itérations. Les préoccupations abordées sont :

- Spécification des données d'adaptation

- Surveillance (*monitoring*) de données d'adaptation depuis
    1. Un fichier XML
    2. Un générateur pseudo-aléatoire
    3. Un fournisseur abstrait
- Spécification du domaine d'adaptation d'un composant
- Substitution de composants (via un composant adaptatif)
- Filtrage des candidats de substitution
- Gestion des instances de candidats de substitution
- Intégration de l'adaptation dans une librairie graphique et son éditeur graphique (Qt et Qt Designer, respectivement)

Les prototypes ont été développés séparément de la librairie de référence.

3.1.4 Développement d'AdaptivePy

AdaptivePy a été développé en deux temps : d'abord dans une phase exploratoire puis dans une phase de formalisation. Pour la réalisation des prototypes, les artéfacts nécessaires à leur réalisation ont été développés dans une version préliminaire d'AdaptivePy. Cette version avait comme but d'être une implémentation rudimentaire et possiblement instable des concepts identifiés puisqu'elle ne comptait pas de tests unitaires.

Une fois les limitations identifiées et les problèmes d'implémentation mieux compris, la librairie a été complètement réécrite depuis les patrons de conception formalisés. Cette nouvelle version inclut des tests unitaires ainsi que de la documentation pour la plupart des classes et méthodes. Ainsi, des tests unitaires ont été configurés sur le site de dépôt en ligne de sorte que ces derniers soient exécutés pour chaque *commit*. De plus, comme la librairie est distribuée publiquement, la configuration pour PyPi (base de données de paquets Python) a été ajoutée.

Le développement de cette librairie a participé à l'analyse des concepts lors du procédé itératif du projet de recherche puisque certaines limitations d'implémentations, autant au niveau de la performance que de la structure imposée par la POO, ont influencé l'organisation des patrons de conception. Les lacunes au niveau des artéfacts n'ont donc pas seulement été identifiées au niveau conceptuel, mais également au niveau pratique. Cette

approche a permis de produire des patrons de conception qui sont plus facilement applicables et liés aux bonnes pratiques de programmation.

3.2 Spécialisation pour les interfaces usagers graphiques

Comme le projet de recherche vise directement le développement des interfaces usagers graphiques, il est important que certaines clarifications soit apportées aux choix effectués pour favoriser ce domaine d'application. Bien que beaucoup d'importance est apportée à la généralisation des patrons de conception, certains systèmes se prêtent moins à la structure produite par l'utilisation des patrons.

Par exemple, le patron Proxy routeur nécessiterait un support explicite de transactions pour des systèmes distribués en réseau pour effectuer l'opération de routage. Bien qu'un transfert d'état est prévu dans ce patron, ce dernier n'a pas été pensé pour supporter la redondance ou le balancement de charge, typiques à ce genre de systèmes. Dans ces applications, il peut exister plusieurs instances de candidats de substitutions issues d'une même classe. Comme ces types d'adaptation ne sont pas rencontrés dans le contexte des interfaces usagers graphiques, les concepts qui y sont reliés ont été laissés de côté. Malgré ce compromis, le patron demeure une bonne base où les concepts manquants peuvent être ajoutés.

Un autre aspect qui est induit par l'application dans le contexte d'interfaces usagers graphiques est la liaison à un *toolkit* comme Qt. En effet, le prototype présenté dans le premier article dépend de certains éléments fournis par Qt et la librairie de *binding* PyQt. Principalement, la solution rendant possibles la création de *widgets* personnalisés et leur chargement dynamique dans l'éditeur graphique Qt Designer est spécifique à ce *toolkit*. Comme chaque *toolkit* a sa propre structure, il aurait été nécessaire de créer une couche supplémentaire d'abstraction pour rendre le prototype de l'étude de cas neutre à ce niveau. De plus, AdaptivePy n'inclut aucun artéfact spécifique aux interfaces graphiques. Ainsi, il a été jugé qu'une telle fonctionnalité aurait compliqué inutilement le prototype présenté. Le compromis choisi a été de simplifier les diagrammes représentant les implémentations (figures 4.5 et 4.7) en enlevant les éléments trop spécifiques à Qt de sorte que l'idée générale derrière soit portable, peu importe le *toolkit* et même hors du domaine des interfaces usagers graphiques.

Il est tout de même adéquat de spécifier l'approche empruntée pour réaliser le composant adaptatif sous forme de widget personnalisé. La figure 3.2 présente la façon dont le prototype démonstratif avec AdaptivePy (voir section 4.5.2) rend visible le *widget* à l'éditeur gra-

phique Qt Designer. Un plug-in fourni par la librairie de *binding* PyQt introspecte le code de l'application en vue d'une classe concrète dérivée de `QPyDesignerCustomWidgetPlugin`. Une telle classe est une classe fabrique qui génère une instance d'un `QWidget`. Dans le cas du prototype, ce *widget* est le composant adaptatif qui supporte la sélection d'options. Qt Designer peut ensuite charger dynamiquement le *widget* lorsque ce dernier est inséré dans l'espace de travail.

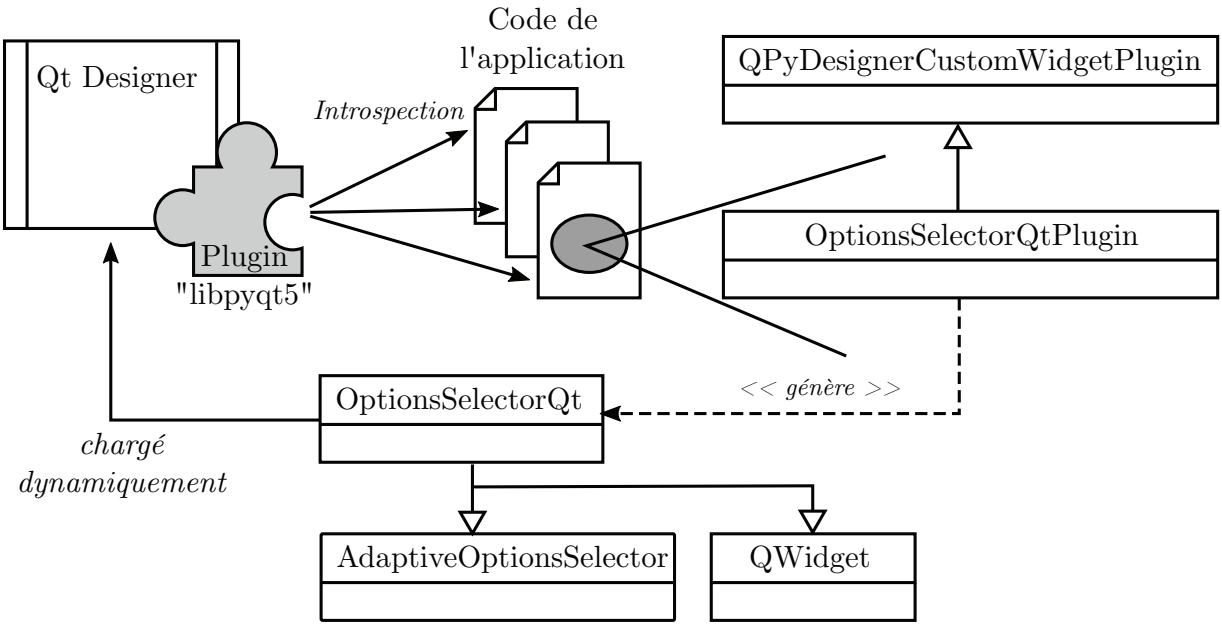

Figure 3.2 Mise en place du composant personnalisé du prototype de la section 4.5.2

3.3 Conclusion

Dans ce chapitre, les étapes qui ont mené à la rédaction d'un premier article dont le sujet est les patrons de conception pour l'ajout de comportements adaptatifs dans les interfaces usagers graphiques ont été présentées.

La méthodologie de recherche des concepts et d'élaboration d'une synthèse des solutions depuis laquelle les patrons de conception ont été élaborés a été présentée. Brièvement, il s'agit d'un procédé itératif où la recherche des concepts est concentrée sur une préoccupation précise de l'adaptation tirée des enquêtes du domaine pendant la période d'une itération. Pendant cette même période, ces concepts ont été appliqués dans des prototypes et, une fois formalisés après plusieurs itérations, implémentés en tant qu'artéfacts dans une librairie de référence nommée AdaptivePy.

Comme les patrons de conception visent directement les interfaces usagers graphiques, certains choix ont été faits pour mieux servir ce domaine d'application. Plus particulièrement, certains types d'adaptation qui ne sont pas rencontrés dans le contexte des interfaces graphiques ont vu leurs concepts spécifiques omis, par exemple dans le cas des systèmes distribués en réseau. Aussi, la neutralité au niveau des *toolkits* n'a pas été considérée comme une fonctionnalité clé, mais les détails spécifiques à Qt ont été omis pour que l'étude de cas soit la plus portable possible tout en restant représentative de l'implémentation réelle.

Le chapitre suivant est une reproduction du premier article introduit dans le présent chapitre. Les concepts retenus sont expliqués en détail, les patrons de conception sont présentés et le prototype développé pour l'étude de cas montre leur utilisation dans un contexte réel. Ce dernier a été réalisé avec la version 0.1.2 d'AdaptivePy telle que distribuée sur PyPi[1].

[1] `https://pypi.python.org/pypi/adaptivepy/0.1.2`

CHAPITRE 4

Article 1 : Design patterns for addition of adaptive behavior in graphical user interfaces

Avant-propos

Auteurs et affiliation :

S. Longchamps : étudiant à la maîtrise, Université de Sherbrooke, Faculté de génie, Département de génie électrique et de génie informatique.

R. Gonzalez-Rubio : professeur, Université de Sherbrooke, Faculté de génie, Département de génie électrique et de génie informatique.

Date d'acceptation : 4 décembre 2016

État de l'acceptation : version finale publiée[1]

Revue : Proceedings of the Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications

Référence : [Longchamps et Gonzalez-Rubio, 2017]

Titre français : Patrons de conception pour l'ajout de comportements adaptatifs dans les interface usagers graphiques

Contribution au document : Cet article présente les trois patrons de conception qui englobent les concepts fondamentaux extraits de la littérature et des *frameworks* existants à ce jour. Cet article vise l'atteinte du premier objectif secondaire qui est la proposition de patrons de conception. Chaque sous-objectif discuté à la section 1.3 est abordé dans l'article. Notamment, l'utilisation d'un cas d'étude sous forme de prototype d'application graphique est utilisé pour démontrer l'utilisation des patrons et comparer le code résultant à une implémentation *ad hoc*. L'utilisation de la librairie de référence développée, AdaptivePy, est également expliqué en détails.

Résumé français : Les interfaces usagers graphiques dans les logiciels modernes requièrent de plus en plus à s'adapter à diverses situations et utilisateurs, ce qui rend leur

[1] `https://www.thinkmind.org/index.php?view=article&articleid=adaptive_2017_1_20_50006`

développement plus complexe. Pour gérer cette complexité, nous présentons dans cet article trois patrons de conceptions, *Moniteur*, *Proxy routeur* et *Composant adaptatif*, en tant que solutions au problème d'implémentation graduelle de comportement adaptatif dans les interfaces usagers graphiques et en général dans tout logiciel basé sur des composants. Plutôt que de proposer de nouveaux mécanismes d'adaptation, notre objectif est de formaliser une structure de base pour l'ajout progressif de différents mécanismes tout au long du cycle de développement. Pour ce faire, les travaux existants sur le sujet des patrons de conception orienté vers l'adaptation ont été explorés et les concepts reliés à des préoccupations similaires sont extraits et généralisés dans de nouveaux patrons. Ces patrons ont été implémentés dans une librairie de référence en langage Python baptisée AdaptivePy, elle-même utilisée dans une étude de cas sous forme d'application graphique. Cette étude de cas montre une utilisation concrète des patrons et est comparée à une implémentation *ad hoc* fonctionnellement équivalente. Nous observons que la séparation des préoccupations est promue par les patrons de conception et le potentiel de testabilité est amélioré. De plus, l'adaptation des composants graphiques peut être prévisualisée depuis l'éditeur graphique de l'interface. Cette approche est plus près du flux de travail standard utilisé dans le développement d'interfaces usagers graphiques et n'est pas réalisable avec la solution *ad hoc*.

Design Patterns for Addition of Adaptive Behavior in Graphical User Interfaces

Samuel Longchamps, Ruben Gonzalez-Rubio

Sherbrooke University,
Sherbrooke, Québec, Canada
Email: {samuel.longchamps, ruben.gonzalez-rubio}@usherbrooke.ca

***Abstract*—Graphical user interfaces (GUI) in modern software are increasingly required to adapt themselves to various situations and users, rendering their development more complex. To handle complexity, we present in this paper three design patterns, *Monitor*, *Proxy router* and *Adaptive component*, as solutions to the gradual implementation of adaptive behavior in GUI and general component-based software. Rather than proposing new adaptation mechanisms, we aim at formalizing a basic structure for progressive addition of different mechanisms throughout the development cycle. To do so, previous work on the subject of design patterns oriented toward adaptation is explored and concepts related to similar concerns are extracted and generalized in the new patterns. These patterns are implemented in a reference Python library called AdaptivePy and used in a GUI application case study. This case study shows concrete usage of the patterns and is compared to a functionally equivalent *ad hoc* implementation. We observe that separation of concerns is promoted by the patterns and testability potential is improved. Moreover, adaptation of widgets can be previewed within a graphical editor. This approach is closer to the standard workflow for GUI development which is not possible with the *ad hoc* solution.**

Keywords–adaptive; design pattern; graphical user interface; context; library.

I. Introduction

As applications become increasingly complex and distributed, adaptive software has become a research subject of great interest to tackle related challenges. One area of modern applications where adaptation requirements have flourished is graphical user interfaces (GUI). Because they are generally engineered using a descriptive language and oriented toward specific platforms, it is hard to produce a single GUI which automatically adapts itself to its multiple usage contexts.

Many researchers have proposed models and frameworks to implement adaptive behavior in a generic manner for components-based software [1]–[4]. These solutions typically require significant effort to modify an existing software architecture and make many technological choices and assumptions. They are limited both in terms of gradual integration to the software and in portability, for a framework usually targets a certain application domain (e.g. distributed client-server systems). As a more portable approach, we propose to use design patterns for formalizing structures of components which can be easily composed to produce specialized adaptive mechanisms. While some work has been done to propose design patterns for the implementation of common adaptive mechanisms [5]–[8], the present work aims at generalizing widespread concepts used in these patterns. In doing so, their integration in existing software is expected to be easier and more predictable.

As a proof-of-concept, a reference implementation of the design patterns has been done as a Python library called AdaptivePy. An application was built as a case study using the library to validate the gains provided by the patterns compared to an *ad hoc* solution. Special attention was paid to the compatibility to modern GUI design workflow. In fact, rather than create a specialized toolkit or create a custom designer tool which would include the design patterns' artifacts, the Qt cross-platform toolkit along with the Qt Designer graphical editor was used. The application workflow is presented and compared to original methods and advantages are highlighted. We expect that through the case study, the patterns' usage and advantages will be clearer and offer hints on how to structure an adaptive GUI.

The remainder of this paper is organized as follows. Fundamental concepts of software adaptation extracted from previous work are described in Section II. The design patterns inspired from the concepts are presented in Section III. The prototype application with adaptive GUI is presented in Section IV and an analysis of the gains procured by the use of the proposed design patterns are presented in Section V. The paper concludes with Section VI and some future work is discussed.

II. Concepts of Software Adaptation

This section presents major concepts of adaptation from related work classified in three concerns: data monitoring, adaptation schemes and adaptation strategies.

A. Adaptation Data Monitoring

Contextual data on which customization control rely, referred to as *adaptation data* in this paper, can come from various sources, both internal (for "self-aware" applications [9]) and external (for "self-situated" [9] or "context-aware" applications). The acquisition of contextual data to be used as adaptation data is part of a primitive level, which is necessary for other more complex adaptation capabilities to be implemented [10]. Contextual data is usually acquired by a monitoring entity (sensors/probes/monitors) responsible for quantizing properties of the physical world or internal state of an application [7], [11]–[15]. Multiple simple sensors can be composed to form a complex sensor, which provide higher-level contextual data (Sensor Factory pattern [15]). Internal contextual data can be acquired simply by using a component's interface, but when the interface does not provide the necessary methods, introspection can be used (Reflective Monitoring

 ISBN: 978-1-61208-532-6

[15]). When a variety of adaptation data is monitored, it provides a modeled view of the software context, sometimes shared within a group of components. Some event-based mechanism with registry entities can be used to propagate adaptation data to interested components (Content-based Routing [15]). Quantization can be done on multiple abstraction levels and thresholds can be used to trigger adaptation events (Adaptation Detector [15]).

B. Adaptation Schemes in Components

Many researchers aimed at defining a design pattern for an adaptive component that would allow for various schemes of adaptation in a generic way. Two main approaches can be extracted from previous work: component substitution and parametric adaptation.

a) ***Component substitution****:* The underlying principle of component substitution is to replace a component by a functionally equivalent one with regard to a certain set of features. This can also be done by adding an indirection level to the dispatching of requests and forwarding them to the appropriate component. The first pattern applying this concept is probably the Virtual Component pattern by Corsaro, Schmidt, Klefstad, *et al.* [5]. It is similar to the adaptive component proposed by Chen, Hiltunen, and Schlichting [16], but adds the principle of dynamic (un)loading of substitution candidates. In both cases, an abstract proxy is used to dispatch requests to a concrete component, which is kept hidden from the client. This approach is also used by Menasce, Sousa, Malek, *et al.* [17], who proposed architectural patterns to improve quality of service on a by-request dispatch to one or many components. To maintain the software in a valid state before, during and after the substitution, many techniques have been proposed, such as transiting a component to a quiescent state [18], [19] and buffering requests [20]. State transfer between components can be used when possible, otherwise the computing job must be restarted [16], [19].

b) ***Parametric adaptation****:* Rather than substituting a whole component by a more appropriate one, parametric adaptation is when a component can adapt itself to be more appropriate to a situation. This is usually done by tuning *knobs*, configurable units in a component (e.g. variables used in a computation). Knobs can be exposed in a *tunability interface* [1] for use by external control components, either included by design or automatically generated at the meta-programming level (e.g. with special language constructs, such as annotations [10]). The tunability domain of each knob is explicit and may vary over time. For example, if a new algorithm is discovered in the middle of a large computing job, an adaptation mechanism that is kept aware of the knob's possible values is able to switch to it if it judges that it will perform better overall [21].

C. Adaptation Strategies

No single adaptation strategy is universal for all software. Most past work has been done on applying component substitution using various strategies. For example, many researchers have explored rule-based constraints along with an optimization engine to devise architectural reconfiguration plans [1], [13]. This popular approach has tainted proposed frameworks that tend to be limited to this strategy only. An important principle is that strategies are separate from the component's implementation and can be easily changed. In fact, it is desirable to externalize adaptation strategies in order to be able to easily develop, modify and test them separately. Ramirez [7] calls this class of design patterns "decision-making", since they relate to when and how adaptation is performed. Because these design patterns are concrete adaptation strategies, their artifacts are mainly related to specific strategies (e.g. inference engines, rules, satisfaction evaluation functions). The approach of this class of patterns is typically related to rule-based constraints solving, but a more general goal is to select which plan or components from a set to reconfigure the system with.

III. Design Patterns

This section presents design patterns which realize the concepts presented in Section II with some improvements. When used together, we believe they provide the sought structure for adaptive software. Unified modeling language (UML) diagrams are used to show the structure of the patterns in a standardized way.

A. Monitor Pattern

Classification: Monitor and analyze.
Intent: A monitor provides a value for one type of adaptation data to interested entities.
Motivation: There is a need to quantize raw contextual data as parameters of adaptation data with explicitly defined domain and in specialized modules decoupled from the rest of the application. Adaptation data needs to be reasoned about in arbitrarily high abstraction level and be proactive in the adaptation detection process. Agreement for monitored data should be implied by design in order to allow for safe information sharing among interacting components.
Structure: Fig. 1 shows the structure of the monitor pattern as a UML diagram.

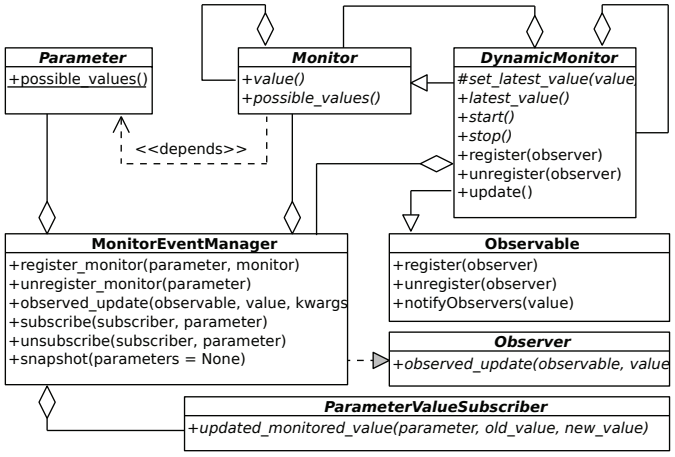

Figure 1. Monitor pattern UML diagram

Participants:

- **Parameter:** A parameter is one type of adaptation data as defined in Section II-A. Its possible values domain is explicitly defined and forms a state space. Many range types can be used to model a parameter's domain.
- **(Static) Monitor:** Provides a stateless (further referred to as "static") means of acquiring a value within

a subset of a certain parameter's domain. Formally, $\Omega_M \subseteq \Omega_P$ for possible values Ω of monitor M and parameter P. A monitor can be an aggregation of other static monitors, but not of dynamic monitors.

- **Dynamic monitor:** Additionally to providing a value for a parameter, schedules the acquisition of the value and alerts an observer that a new value has been acquired. Some form of polling or interrupt-based thread awakening needs to be employed along with a previous value to know if the value has changed compared to the latest value, in which case an event notification is triggered to interested entities. This makes it inherently stateful. Like a static monitor, it can be an aggregation of other monitors. The particularity is that it can aggregate both static and dynamic monitors.
- **Monitor event manager:** Registry entity which allows for a client component to subscribe to a parameter and be alerted when a new value is acquired. Similarly, a dynamic monitor can be registered within the manager and provide a value to any subscriber of the corresponding parameter. In such manager, monitors and parameters are related by a one-to-one relationship; a given parameter can only be monitored by a single monitor.
- **Observable/Observer:** See Gang of Four observer pattern [22]. Used for monitor registering mechanism.
- **Parameter value subscriber:** Provides a means to be notified when a new value of a parameter it has subscribed to has been acquired.

Behavior: An adaptation data type can be formalized as a parameter in terms of the quantized values the system expects to use. A static monitor provides a means to concretely quantize raw contextual data from a sensor or introspection to a value within a defined domain expected by the system. The quantization can be done using fixed or variable thresholds. A dynamic monitor adds scheduling behavior, which allows to provide a value based on accumulated data over time and apply filtering. The monitor event manager is alerted by monitors and dispatches the new value to related subscribers. The dependency regarding subscribers is with the parameters for which they requested to be notified, but actual monitoring is done separately.
Consequences: As monitors are hierarchically built, higher-level abstraction information can be provided. This pattern allows the analysis step of a MAPE-K loop [12] to be done through hierarchical construction of monitors: a parameter can define high-level domain values which are provided by a monitor composed from lower-level ones and components can use this to simplify their adaptation strategies. High-level adaptivity logic is reusable in that parameters are abstract and can easily be shared among projects. Monitors can be chained such that only the concrete data acquisition has to be redone between projects, keeping scheduling and filtering as reusable entities.
Constraints: To assure agreement between interacting components, it is necessary for adaptive components which depend on a common parameter to also subscribe to the same monitor event manager. These components are therefore part of the same *monitoring group*. This can be checked statically or be assumed by contract. The need for a one-to-one relationship between a monitor and a parameter within a monitoring group is based on this agreement requirement. A monitoring group can be thought of as a single entity that cannot have duplicate or contradicting attributes, e.g., it cannot be at two positions at once. In this example, an attribute is a parameter and a monitor is the entity providing the value for this attribute.
Related patterns: Sensor factory, reflective monitoring, content-based routing, adaptation detector [7], information sharing, observer [22].

B. Proxy Router Pattern

Classification: Plan and execute.
Intent: A proxy router allows to route calls of a proxy to a component chosen among a set of candidates using a designated strategy.
Motivation: When implementing component substitution, a way to clearly separate concerns relating to the adaptation logic (substitution by which component) and the execution of substitution (replacing a component or forwarding calls to it) are difficult to implement in an extensible way. The proxy pattern [22] allows to forward calls to a designated instance, but does not specify how control of the routing process should be implemented. Candidate components need to be specified in a way that does not necessitate immediate loading or instantiation and which is mutable (to allow runtime discovery). To maximize reusability, strategies should be devised externally.
Structure: Fig. 2 shows the structure of the proxy router pattern as a UML diagram.

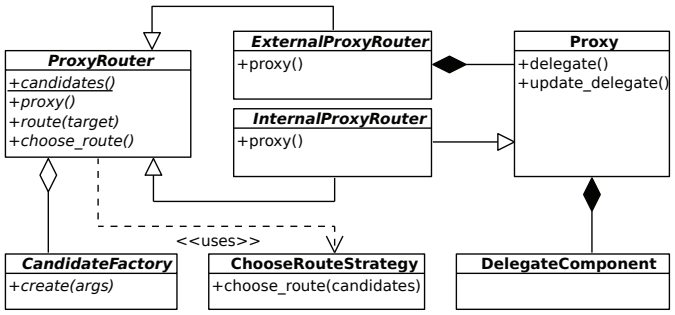

Figure 2. Proxy router pattern UML diagram

Participants:

- **Proxy:** Gang of Four [22] proxy pattern, with the exception that the interface is not necessarily specified (e.g. forwarding to introspected methods). It is responsible for making sure no calls are lost when a new delegate is set.
- **Delegate component:** Concrete component which is proxied. It must be specified as part of the proxy router's candidates set.
- **Proxy router:** Keeps a set of component candidates and allows to control the routing of the calls a proxy receives to the appropriate candidate chosen by some strategy. The proxy router is responsible for ensuring any state transfer and initialization of candidate instances.
- **Candidate factory:** Gang of Four [22] factory pattern for a candidate. Used as part of candidates definition.

 ISBN: 978-1-61208-532-6

Can do local loading/unloading for external candidates.

- **Choose route strategy:** Concrete strategy to choose which candidate among a set to use, based on Gang of Four [22] strategy pattern. It uses accessible information from the application, candidates (e.g. adaptation space, descriptor, static methods) or any inference engine available to make a choice.
- **External/Internal proxy router:** Depending on the use, a proxy router can *use* an external proxy (as a member) or internally *be* a proxy (through inheritance). To allow for both schemes, a means to acquire the proxy is provided and returns either the member object (external) or a reference to the proxy router itself (internal).

Behavior: A set of candidates is either statically specified or discovered at runtime (e.g. looking for libraries providing candidates). The proxy router is then initialized by choosing a candidate using the strategy and controls the proxy to set an instance of the chosen candidate as active delegate. At any time, a new candidate can be chosen and set as active delegate of the proxy.
Consequences: The proxy router pattern allows for flexible and extensible specification of component substitution. The strategies to choose a candidate to route to can be reused in any project with consistent information acquisition infrastructure, such as the one provided by the monitor pattern. Candidates need not be specified statically and control related to routing can be done both internally and externally.
Constraints: Strategies might rely on certain project specific information which is not portable. Separating specific from generally applicable strategies and composing them should help with this constraint.
Related patterns: Adaptive component [16], virtual component [5], master-slave [23], component insertion/removal, server reconfiguration [7], proxy [22].

C. Adaptive Component Pattern

Classification: Analyze and plan.
Intent: Use monitored adaptation data to control parametric adaptation and component substitution by making adaptation spaces explicit.
Motivation: A basic structure is needed to easily add adaptive behavior in the form of parametrization or substitution. Components need a way to explicitly provide means for adaptation strategies to reason about their adaptation space in order to formulate plans. This information should be external to a base component if the adaptation is to be added gradually. Most importantly, an adaptive component must behave like any non-adaptive component and be used among them without any impact on the rest of the system.
Participants:

- **Adaptive:** An adaptive component which defines means for acquiring the adaptation space. It can be used as a subscriber to a parameter value provider.
- **Monitor event manager:** Parameter value provider realized with the monitor pattern (see Section III-A).
- **Parameter value subscriber:** Provides a means to be notified when a new value of a parameter it has subscribed to has been acquired (see Section III-A).
- **Proxy router:** Proxy router pattern (see Section III-B)
- **Adaptive proxy router:** Adaptive version of a proxy router allowing to drive the routing process (substitution) using monitored data.

Structure: Fig. 3 shows the structure of the adaptive component pattern as a UML diagram.

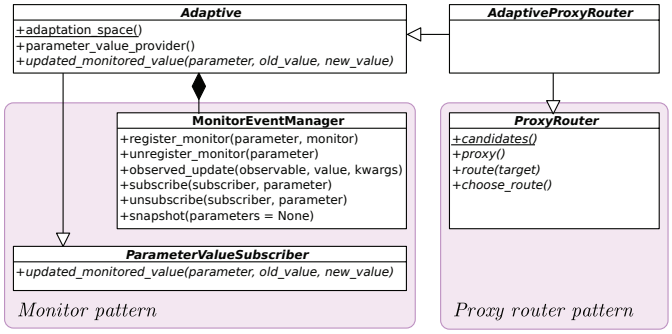

Figure 3. Adaptive component pattern UML diagram

Behavior: A component to be made adaptive can inherit the adaptive interface or a specific decorator can be created if the component's code should remain unchanged. The adaptive implementation defines what base adaptation space it will support. Then, knobs can be defined within the component and used as variables to compute, for example, its size or lay outing specifications. Tuning can be done when an updated parameter value is received. For substitution, the process is the same, but uses the AdaptiveProxyRouter interface. Specific strategies can be created, using as many generic filters as possible (e.g. filter out candidates which adaptation space does not overlap with a snapshot of the current state).
Consequences: Because of the explicit declaration of adaptation space, strategies can easily reason about how a component can behave in a situation. For example, a strategy can use the fact that a component's space is too specific or too wide. Any component can be made adaptive and does not require modifications to other components. Because of the support for both parametric adaptation and component substitution, the basic structure proposed in this pattern is suitable for virtually any adaptive mechanism based on monitored data and components adaptation spaces.
Constraints: Like stated in Section III-A, interacting adaptive components must subscribe to the same monitor event manager to assure consistency in decision-making processes. While arbitrarily large hierarchies of adaptive components can be composed, there is an inherent overhead induced in the adaptation and routing process. Because a component subscribing to some parameter value provider such as the monitor event manager has no guarantee that this parameter is being actively monitored, adaptive components need to define a default behavior or immediately request a snapshot of the current state. To minimize this effect, it is preferable to register monitors prior to creating any adaptive component.
Related patterns: Monitor (III-A), proxy router (III-B), adaptive component [16], virtual component [5].

IV. Prototype

This section presents AdaptivePy, a reference library implementing the three design patterns presented in this paper,

along with a prototype as a case study for analyzing the gains they procure compared to an *ad hoc* implementation.

A. AdaptivePy

AdaptivePy implements artifacts from all three design patterns described in this paper. The Python language was chosen because it is reflective, dynamically typed and many toolkit bindings are freely available. Beyond the patterns, AdaptivePy provides some useful implementations, such as enum-based discrete-value parameters, push/pull dynamic monitor decorators, operations over adaptation spaces (extend, union, filter) and an adaptation strategy based on substitution candidates' adaptation space restrictiveness. The library is freely available from the PyPi repository under the name "adaptivepy" and is distributed under LiLiQ-P v1.1 license.

B. Case Study Application

The case study application is a special poll designed to favor polarization. Five yes/no questions are asked to a user and answered by selecting the most appropriate response among a list of options. The options provided include yes, no, mostly yes, mostly no and 50/50. To favor polarization, statistics from the previous answers are used to restrict the range of options provided to the user. If the polarization is judged insufficient because of mixed responses (low polarization), fewer options are provided. On the contrary, if virtually all users have answered yes (high polarization), more options in between will be given. The workflow of the application is to start the "quiz" using a Start button, choose appropriate options and send the form using a Submit button. If some options remain unselected, a prompt alerting the user is shown and the form can be submitted again once all options are selected.

The adaptation type used is a form of *alternative elements* [24]. The GUI is made plastic by replacing control widgets displaying the available options at runtime, conserving the option selection feature in any resulting interface. Because there is a varied number of options, some widgets are more appropriate than others to display them, while some cannot display certain amounts of options. A checkbox can handle two options, radio buttons could be used for ranges of two to four options and a combo box for five and more options. Of course, radio buttons can hold more options and the combo box less, but the amounts suggested represent the ranges they are subjectively considered most appropriate for. These can be chosen by a designer and further refined through user testing, which means they must be easy to edit.

Polarization levels act as adaptation data to drive adaptation. An appropriate solution would allow to design the GUI within Qt's graphical editor "Qt Designer" and to preview of the adaptation directly, rather than having to add the business logic beforehand. It would also allow for gradual addition and modification of control widget types without necessitating changes in unaffected modules.

The toolkit used for this application is Qt 5 through the PyQt5 wrapper library. It is a cross-platform toolkit library which provides implementations of widgets like checkboxes, combo boxes are radio buttons groups. The concrete work is therefore limited to implementing how these components can replace each other at the appropriate time and how they are included in a main user interface. We are therefore more interested in the underlying structure of adaptation within the application than specific adaptation strategies and their user-perceived effectiveness. Once an appropriate structure is in place, we expect these can be more easily devised, tested and improved.

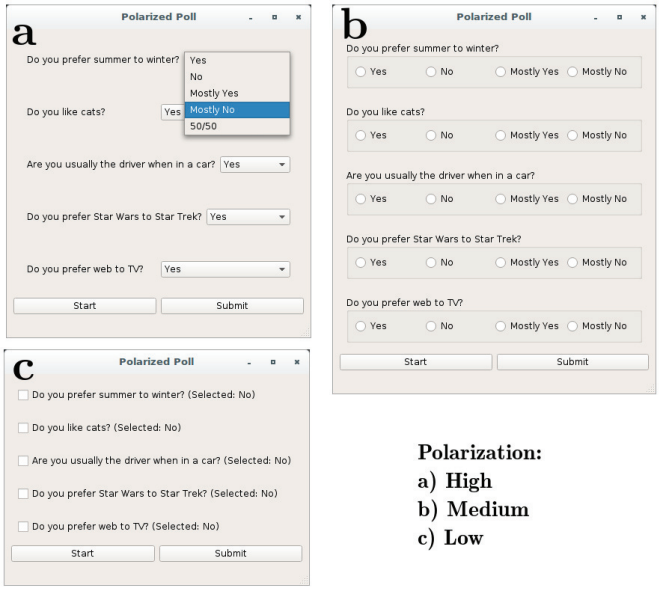

Figure 4. Adaptive case study application "Polarized Poll"

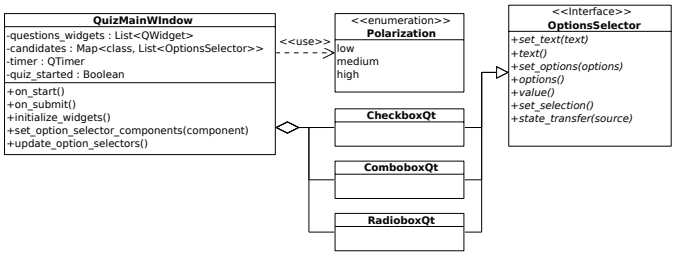

Figure 5. Simplified UML diagram of *ad hoc* implementation of case study application

V. Results

The windows shown on Fig. 4 are the resulted GUI for the application in all three polarization states. Because this case study's focus is on GUI, the monitoring of past responses was simulated and a random monitor is used instead which updates its value by means of a polling dynamic monitor every second, allowing to easily observe adaptation.

A. Ad hoc *Application*

A simplified UML diagram of the *ad hoc* implementation is shown on Fig. 5. The chosen approach is to add placeholder widgets in QuizMainWindow which will be substituted by an appropriate component instance at runtime: CheckboxQt, ComboboxQt or RadioboxQt. A polarization level defined in the enum Polarization is bound to each of these types. A timer within QuizMainWindow polls the polarization value and calls `set_options_selector_components` with the appropriate type. Adaptation control, along with any customization necessary, is entirely done in QuizMainWindow.

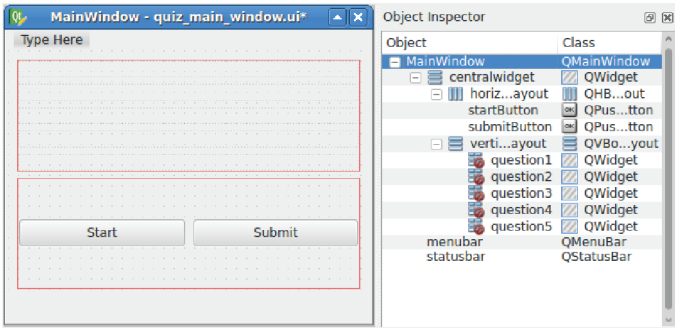

Figure 6. Qt Designer using plain widgets as placeholder for *ad hoc* implementation

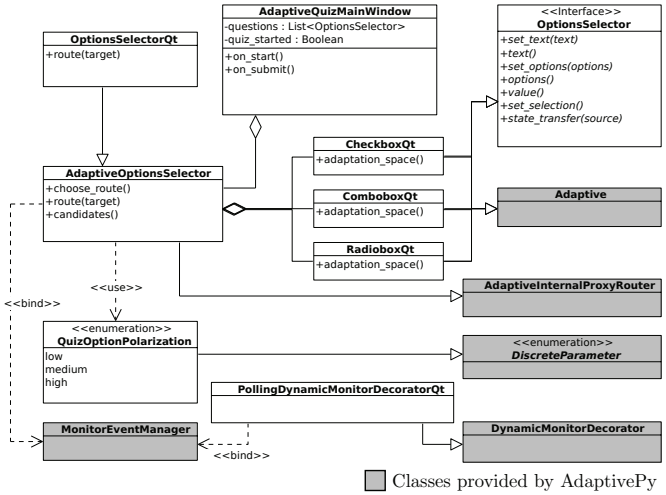

Figure 7. Simplified UML diagram of case study application implementation using AdaptivePy

Fig. 6 shows Qt Designer as the main window is created for the *ad hoc* implementation. Notice that because placeholder components are blank, no feedback is given to the designer. It is therefore not possible to test the controls or set the question label. This makes the approach incompatible with the usual GUI design workflow, which involves previewing the application in the graphical editor before adding business logic.

When analyzing the *ad hoc* code, it is obvious that separation of concerns is not respected since the option selection logic is tangled to its owner element, the main window. Concerns such as scheduling for recomputing polarization and component substitution are mixed with GUI setup and handling of the business flow. This leads to a lack of extensibility, a tangling of concerns and limits unit testing of components. A method is used to select which control component to use based on the polarization, but this solution remains inflexible. The knowledge of adaptation is hidden and cannot be used to devise portable strategies.

One of our goals is to gradually add adaptation mechanisms to GUI implementations, but this is difficult since modification of important classes will add risk of introducing defects. Also, there is no easy way to work on adaptation mechanisms separately from the application. In fact, we cannot separately test the adaptation logic and integrate it after. Generally, the lack of cohesion induced by the inadequate separation of concerns is a sign of low code quality. Because no adaptation mechanism can easily be introduced, modified and reused in other projects, the *ad hoc* implementation works for its specific application case, but is subject to major efforts in refactoring when requirements and features will be added throughout its development cycle.

B. Application Using AdaptivePy

A simplified UML diagram of the application is shown on Fig. 7. From it, we see that the polarization is a discrete parameter and is used by AdaptiveOptionsSelector, specifically to define its adaptation space based on the ones provided by its substitution candidates: CheckboxQt, ComboboxQt and RadioboxQt. Additionally to adaptation by substitution, RadioboxQt can parametrically adapt to changes of polarization levels {low, medium}, since they respectively correspond to 2 and 4 options. Its behavior is that the appropriate number of options is shown depending on the polarization level. AdaptiveQuizMainWindow is free of adaptation implementation details and simply uses the AdaptiveOptionsSelector instances as a normal OptionsSelector. OptionsSelectorQt is a subclass to AdaptiveOptionsSelector which is used as a graphical proxy to candidate widgets. It also defines properties used in Qt's graphical editor Qt Designer, in this case the question label.

Every AdaptiveOptionsSelector instance is made a subscriber to the QuizOptionPolarization parameter at initialization. They are updated when a change in the monitored value is detected, i.e., when a monitor detects a value is different from the previous one. This is because identical subsequent parameter values are expected by default to lead to the same state, so they are filtered out. In the case of AdaptiveOptionsSelector, because it is a proxy router, `choose_route` is called to determine which substitution candidate to route to. Prior to using an adaptation strategy to select the most appropriate candidate, inappropriate ones can be filtered out using `filter_by_adaptation_space`. This function, provided by AdaptivePy, takes a list of candidates along with a snapshot of the current monitoring state and only returns those with adaptation space supporting the current context. Then, a strategy like `choose_most_restricted` is used to choose among valid components. If no component is valid, an exception is raised. With a candidate chosen, all that remains is configuring the proxy router by calling the `route` method with the chosen candidate. This method must also take care of state transfer between the previous and new proxied components. This feature is already defined in the common interface OptionsSelector as `state_transfer`.

Fig. 8 shows Qt Designer as the main window is created with the AdaptivePy-based implementation. When compared to Fig. 6, we notice that the designer has a full view of how the application will look. Moreover, the currently displayed adaptation can be controlled through the setup of the monitors. For example, it is possible to replace the random value by one acquired from a configuration file and trigger adaptation manually. Also, each question is simply a OptionsSelectorQt component rather than a placeholder component and the question is entered directly from the graphical editor using the label property (bottom-right). A major advantage is that adaptive

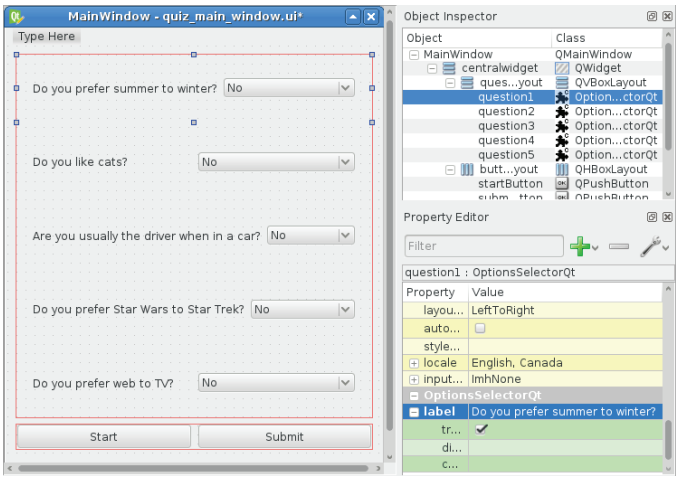

Figure 8. Qt Designer using adaptive components developed with AdaptivePy

components can be reused in other interfaces because they are provided as standalone components. The need for easy edition of adaptation spaces is also addressed by modifying or overriding the `adaptation_space` method of adaptive components.

The adaptation logic is essentially located in the adaptive proxy router class: AdaptiveOptionsSelector. Because adaptation is separated from the rest of the business logic, the main window class can use the adaptive components without the knowledge of adaptation. The only logic remaining is with regard to buttons handling (Start and Submit buttons). It is clear in this implementation that the knowledge of adaptation space which was hidden in the *ad hoc* implementation is used to efficiently choose a substitution candidate. Self-healing action such as replacing a failing component can be easily realized by monitoring the components and including this logic as a strategy. This is not easily realizable in the *ad hoc* implementation. In the prototype, a radio box could safely replace a checkbox since it parametrically covers its full adaptation space, overlapping on {low} polarization. Also, from this case study, we can see that arbitrarily large hierarchies of adaptive and non-adaptive components can be built without tangling code or affecting other components when adding new adaptive behavior.

VI. Conclusion and Future Work

Design patterns presented in this paper can be used as a basic structure to accomplish various levels of adaptation in GUI. Adaptive components can be used with other modules such as recommendation engines to provide more or less automation and proactive adaptation. Monitors can also be extended and even implemented as adaptive components themselves, relying on other more primitive monitors. Proxy routers allow to simplify hierarchical development of arbitrarily large sequences of component substitutions. The patterns form together an effective approach for the integration of various adaptation mechanisms and, in the case of GUI, can be used to provide a more usual workflow than the *ad hoc* implementation. AdaptivePy, as a reference library, is an example of the viability of the patterns when used in a concrete implementation. Even though a simple application was used to observe gains, the solution is applicable to more complex scenarios where multiple parameters, monitoring groups and large hierarchies of adaptive components. The patterns are general enough that they can be used for adding adaptive behavior based on user, environment and platform variations.

Future work will focus on exploring parameters types with more complex value domains and try to formalize a structure to express them. Also, the lack of adaptation quality metrics for verification and validation methods limits the evaluation of gains. To alleviate this limitation, new metrics using concepts of the design patterns presented in this paper will be explored. The goal is to better quantify the quality level of prototypes with regard to adaptation.

References

[1] F. Chang and V. Karamcheti, "A framework for automatic adaptation of tunable distributed applications," *Cluster Computing*, vol. 4, no. 1, pp. 49–62, 2001, ISSN: 1573-7543.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[3] Y. Maurel, A. Diaconescu, and P. Lalanda, "Ceylon: A service-oriented framework for building autonomic managers," in *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, Mar. 2010, pp. 3–11.

[4] M. Peissner, A. Schuller, and D. Spath, "A design patterns approach to adaptive user interfaces for users with special needs," in *Proceedings of the 14th International Conference on Human-computer Interaction: Design and Development Approaches - Volume Part I*, ser. HCII'11, Orlando, FL: Springer-Verlag, 2011, pp. 268–277.

[5] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual component - a design pattern for memory-constrained embedded applications," in *In Proceedings of the Ninth Conference on Pattern Language of Programs (PLoP*, 2002.

[6] G. Rossi, S. Gordillo, and F. Lyardet, "Design patterns for context-aware adaptation," in *2005 Symposium on Applications and the Internet Workshops (SAINT 2005 Workshops)*, Jan. 2005, pp. 170–173.

[7] A. J. Ramirez, "Design patterns for developing dynamically adaptive systems," Master's thesis, Michigan State University, 2008.

[8] T. Holvoet, D. Weyns, and P. Valckenaers, "Patterns of delegate mas," in *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Sep. 2009, pp. 1–9.

[9] M. G. Hinchey and R. Sterritt, "Self-managing software," *Computer*, vol. 39, no. 2, pp. 107–109, 2006.

[10] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[11] M. L. Berkane, L. Seinturier, and M. Boufaida, "Using variability modelling and design patterns for self-adaptive system engineering: Application to smart-home," *Int. J. Web Eng. Technol.*, vol. 10, no. 1, pp. 65–93, May 2015, ISSN: 1476-1289.

[12] IBM, "An architectural blueprint for autonomic computing," IBM Corporation, Tech. Rep., 2005.

[13] S. Malek, N. Beckman, M. Mikic-Rakic, and N. Medvidovic, "A framework for ensuring and improving dependability in highly distributed systems," in *Architecting Dependable Systems III*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 173–193.

[14] V. Mannava and T. Ramesh, "Multimodal pattern-oriented software architecture for self-optimization and self-configuration in autonomic computing system using multi objective evolutionary algorithms," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ser. ICACCI '12, Chennai, India: ACM, 2012, pp. 1236–1243.

[15] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ACM, 2010, pp. 49–58.

[16] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Distributed Computing Systems, 2001. 21st International Conference on.*, Apr. 2001, pp. 635–643.

[17] D. A. Menasce, J. P. Sousa, S. Malek, and H. Gomaa, "Qos architectural patterns for self-architecting software systems," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10, Washington, DC, USA: ACM, 2010, pp. 195–204.

[18] H. Liu and M. Parashar, "Accord: A programming framework for autonomic applications," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 36, no. 3, pp. 341–352, May 2006, ISSN: 1094-6977.

[19] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: ACM, 2006, pp. 371–380.

[20] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, Sierre, Switzerland: ACM, 2010, pp. 462–469.

[21] P. Kang, M. Heffner, J. Mukherjee, N. Ramakrishnan, S. Varadarajan, C. Ribbens, and D. K. Tafti, "The adaptive code kitchen: Flexible tools for dynamic application composition," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–8.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," in *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, Jun. 2004, pp. 79–88.

[24] M. Bezold and W. Minker, *Adaptive multimodal interactive systems*. Springer Science & Business Media, 2011.

 ISBN: 978-1-61208-532-6

CHAPITRE 5

Développement de métriques d'évaluation de la qualité dans les logiciels adaptatifs

Dans le premier article, il a été question de patrons de conception pour l'ajout de comportement adaptatif dans les interfaces usagers graphiques. Cependant, il a été relevé qu'un manque existait pour qualifier l'implémentation effectuée avec les patrons de conception de « supérieure ». Bien qu'une implémentation avec les patrons semble qualitativement meilleure, il n'est pas clair par rapport à quelles mesures quantitatives cette dernière permet d'atteindre une qualité supérieure et surtout d'assurer que ce qui a été développé supporte bien les cas requis.

Ainsi, le sujet de la deuxième partie du présent projet de recherche est la proposition de métriques pour évaluer une implémentation adaptative. Il a été proposé de développer des métriques pour mesurer la qualité de la structure induite par l'application des patrons de conception. Ce deuxième article, reproduit au chapitre 6, a comme but de présenter de la façon la plus complète possible les nouvelles métriques proposées. Le présent chapitre vise plutôt à présenter un portrait plus large de la V&V liée à la structure induite par les patrons de conception. D'abord, un ensemble de métriques potentielles est présenté. Puis, des détails additionnels concernant le développement de la métrique choisie sont présentés en vue de compléter la description faite dans le deuxième article.

5.1 Recherche des métriques

La structure de base que procure l'application des patrons de conception présentés à la section 4.3 permet l'atteinte de certaines qualités du logiciel comme la séparation des préoccupations, l'extensibilité et la réutilisation. Cependant, l'implémentation faite par les développeurs peut varier d'une application à une autre et certains éléments peuvent induire une mauvaise qualité à l'égard de certaines préoccupations.

Un premier travail a été d'identifier les métriques de logiciels traditionnels et vérifier si des analogies peuvent être faites dans le contexte de logiciel adaptatif. Les conclusions de ce travail sont présentées à la section 5.1.1. Par la suite, la structure elle-même a été analysée pour déceler les limitations de l'implémentation qui ne sont pas évidentes pour

les développeurs. Trois métriques ont été identifiées. La première est la *profondeur de l'arbre de substitution* qui est présentée à la section 5.1.2, la deuxième est la *stabilité de l'adaptation* qui est présentée à la section 5.1.3 et finalement la *couverture de l'espace d'adaptation* qui est présentée à la section 5.1.4. Cette dernière métrique est celle qui a été retenue et qui constitue le sujet du deuxième article.

5.1.1 Métriques de la littérature

L'enquête sur l'évaluation de la qualité dans les systèmes adaptatifs de Raibulet *et al.* [38] est la plus récente qui tente de réunir les métriques connues à ce jour. La constatation principale qui émerge de l'analyse des métriques réunies dans cette enquête est la subjectivité qui est nécessaire pour le calcul des métriques. Par exemple, plusieurs métriques visent à qualifier le *niveau* d'adaptation d'un système en vue de le comparer à d'autres. Il est alors inféré qu'une présence forte de l'adaptation implique une meilleure qualité puisque cette dernière permet d'atteindre des buts d'adaptation comme la tolérance aux fautes ou la reconnaissance de comportement problématique. Cependant, une qualification subjective des composants du logiciel est souvent utilisée. Ainsi, un développeur doit juger si un composant est considéré « adaptatif » et de la granularité des composants au niveau du système.

Dans d'autres cas, il est cependant question d'un nombre de reconfigurations ou de temps passé à effectuer des opérations d'adaptation qui peuvent être déterminées de façon objective. Une approche hybride qui modélise une certaine subjectivité sous forme objective est le calcul du bénéfice ou du coût d'une adaptation : il est alors nécessaire de mesurer les éléments de performance ou de qualité qui peuvent indiquer si le plan d'adaptation mis en place a eu un impact positif ou négatif sur le système. Comme il est nécessaire de spécifier ces éléments manuellement et que la relation d'importance entre ces derniers peut être spécifique à une application et à son domaine, la métrique n'est cependant pas entièrement objective.

Le plus grand constat de la situation actuelle est l'absence de métriques permettant de vérifier si la structure est, indépendamment des mécanismes d'adaptation utilisés, de bonne qualité. Plutôt, les travaux existants focalisent sur la quantification de l'atteinte de buts d'adaptation et de la performance d'un système, peu importe la structure mise en place supportant l'adaptation. Cela permet de comparer deux *systèmes*, mais pas les *mécanismes* de façon indépendants. Il est donc adéquat de proposer de nouvelles métriques pour essayer de mieux découpler ces éléments et surtout d'effectuer l'assurance qualité de la structure

permettant l'adaptation. Pour une analyse plus en profondeur des travaux analysés dans l'enquête de Raibulet *et al.* [38], voir la section 6.2.2.

5.1.2 Profondeur de l'arbre de substitution

Une métrique connue des logiciels traditionnels est la profondeur de l'arbre d'héritage (*depth of inheritance tree*). Cette métrique mesure le chemin d'héritage le plus long dans un système donné. Plus le chemin est long, plus la complexité de l'arbre d'héritage est grande, mais plus le potentiel de réutilisation est élevé, et inversement pour un chemin court. Une métrique similaire peut être identifiée pour les composants adaptatifs. En effet, ces derniers ayant des candidats de substitution, il est possible de mesurer une distance entre un composant adaptatif qui constitue la racine d'un arbre et le candidat de substitution le plus loin y étant reliée.

Par exemple, pour la situation simple où un composant adaptatif a M candidats de substitution, la distance est 1 puisqu'il n'existe qu'un niveau de substitution. Cependant, si l'un d'entre eux a à son tour N candidats de substitution, on calcule que la distance est 2, car il existe un niveau additionnel. Un exemple d'arbre est présenté à la figure 5.1. Dans ce cas, la profondeur est 2 pour le composant L_0. Si on calcule la profondeur de L_{00}, il s'agirait alors de 1 puisqu'un seul niveau de substitution existe sous ce composant. Comme L_{01} et L_{02} n'ont aucun candidat de substitution, la profondeur pour ces composants est 0.

Les implications d'une telle métrique sont plus que de contrôler le niveau de complexité. En effet, plus de niveaux de substitution impliquent un plus grand nombre de stratégies à utiliser pour effectuer les adaptations en cascade. Ainsi, tout dépendant de la complexité de performance des stratégies, il est possible qu'un nombre de niveaux trop élevé de substitution induise un délai trop grand pour être viable. Ainsi, cette métrique pourrait s'avérer utile pour mesurer à quel point une hiérarchie de composants est complexe. Aussi, elle pourrait permettre de déterminer une cause de basse performance dans une application donnée et avec les stratégies utilisées.

5.1.3 Stabilité de l'adaptation

Les composants adaptatifs peuvent réagir à des changements dans les données d'adaptation. Ces changements sont détectés par des moniteurs qui émettent une notification. Cette dernière est distribuée aux composants intéressés par un gestionnaire d'évènements. Cependant, des données d'adaptation peuvent être issues des composants eux-mêmes du logiciel, et donc lorsqu'un composant s'adapte, il est possible que les changements in-

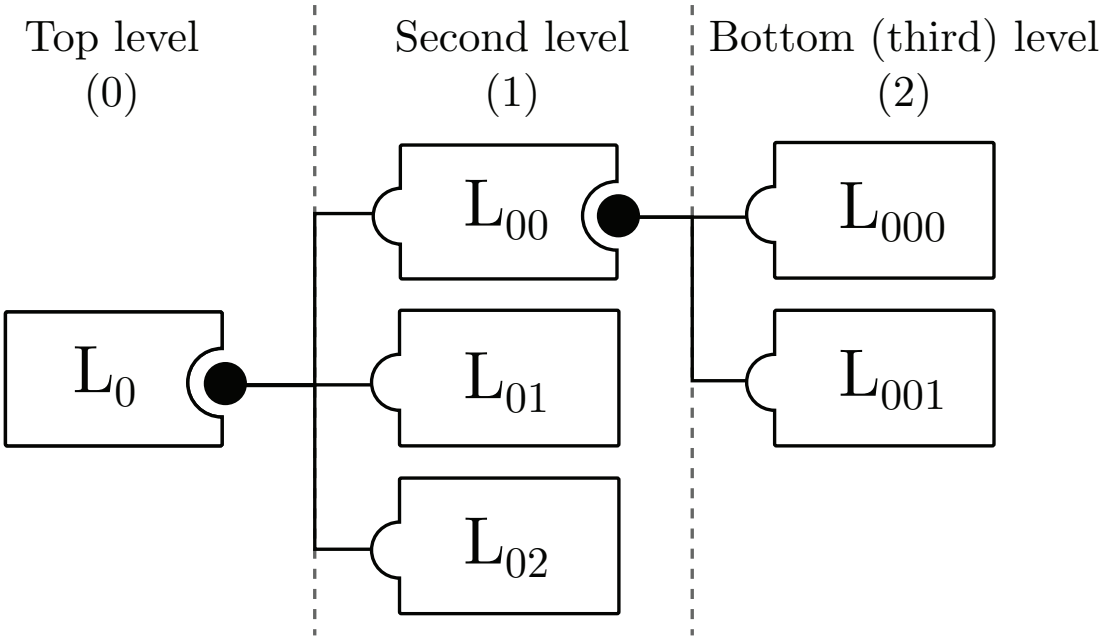

Figure 5.1 Arbre de substitution de composants adaptatifs

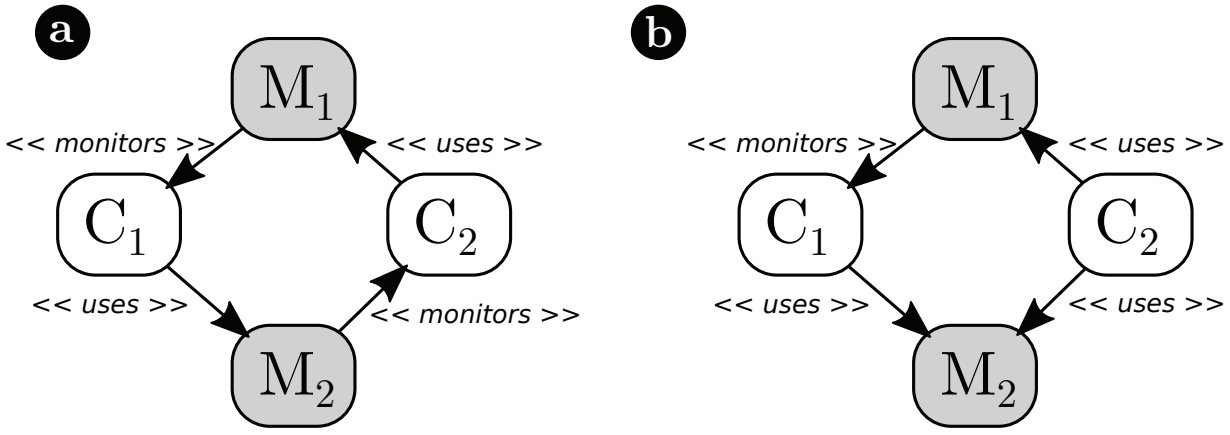

Figure 5.2 Exemples des relations de dépendances entre moniteurs et composants *a)* Boucle de dépendances *b)* Dépendances sans boucle

duisent un changement dans une autre donnée d'adaptation. Dans un cas idéal, cette chaîne d'évènements se stabilise éventuellement et le système ne s'adapte pas continuellement en réaction à une adaptation précédente. Dans un cas problématique, un cycle de réaction d'adaptation existe et le système ne se stabilise jamais. Ainsi, il est nécessaire de prévoir une façon de mesurer la stabilité d'adaptation d'un système en fonction de la façon dont les adaptations peuvent impacter les données d'adaptation.

Une façon de pallier ce problème est de détecter les boucles dans un graphe directionnel où les noeuds sont les composants et leurs moniteurs et où les liens sont les relations de dépendances. Un moniteur dépend d'un composant s'il utilise l'état du composant pour produire une valeur. Un composant dépend d'un moniteur s'il est abonné au paramètre auquel le moniteur est rattaché. Le nombre de boucles indique donc à quel point le système peut être instable. En principe, il serait préférable d'anéantir toute boucle du système (valeur idéale = 0) afin que l'adaptation ne puisse jamais être instable. Cependant, en pratique, certaines boucles ne peuvent être retirées et il s'agit alors d'assurer la stabilité par d'autres moyens.

La figure 5.2 montre deux exemples de graphes de dépendances entre les composants et les moniteurs. En *a)*, on voit qu'une boucle existe pour $\{M_1, C_1, M_2, C_2\}$. Cependant, en *b)*, cette boucle n'est pas présente, même si un lien de dépendance existe entre chacun des éléments. En effet, un changement à C_1 pourrait avoir un impact sur la valeur fournie par M_1, puis sur C_2 puisque ce dernier l'utilise. Cependant, la chaîne ne continuerait pas plus loin, car aucun élément ne dépend de l'état de C_2.

5.1.4 Couverture de l'espace d'adaptation

Un composant adaptatif a comme requis d'exposer le domaine qu'il supporte pour chacun des paramètres qu'il connait. Si chaque domaine est modélisé comme une dimension d'un espace d'états, on peut donc identifier une zone dans cet espace que le composant supporte et qui est appelé espace d'adaptation. L'espace contextuel, soit l'espace d'états dans lequel le système peut être au niveau contextuel, est l'ensemble des états définis pour chacune des dimensions. Ces dimensions sont définies soit à la conception par les paramètres, soit à l'exécution par les moniteurs. La proportion de l'espace contextuel connu d'un composant que couvre son espace d'adaptation est un indicateur de l'étendue de son support pour la variabilité du système. Une couverture $C < 1.0$ implique donc qu'une proportion $1.0 - C$ d'états induit un comportement indéfini lorsque le composant en question est utilisé.

Les candidats de substitution d'un composant adaptatif ayant chacun leur espace d'adaptation, on peut déduire que l'espace d'adaptation du composant adaptatif est par défaut l'union de tous les états supportés par l'un ou l'autre des candidats. Ainsi, il est possible d'obtenir une couverture $C = 1.0$ si les espaces d'adaptation de chacun des candidats de substitution se complémentent. En d'autres mots, si l'union des espaces d'adaptations est égale à l'espace contextuel du système, la couverture sera totale ($C = 1.0$).

5.1.5 Résumé

L'objectif de chacune des métriques potentielles qui ont été proposées dans les sections précédentes était de quantifier un niveau de qualité par rapport à une propriété du logiciel induite par la structure en place. Ces métriques sont résumées ici :

- La **profondeur de l'arbre de substitution** permet de quantifier la complexité de l'architecture de composition des composants adaptatifs. Utilisé conjointement à une analyse de complexité algorithmique des stratégies d'adaptation, il serait possible de quantifier la performance de la mise en place d'une adaptation. Ce genre d'analyse demanderait d'exprimer sous forme de notation asymptotique la complexité de stratégies et de les joindre suivant les liens de composition.

- La **stabilité de l'adaptation** permet d'indiquer si des cycles existent au niveau des dépendances entre les composants et les moniteurs. Un cycle induit une suite d'adaptations possiblement sans fin, ce qui est indésirable et est qualifié « d'instable ». Une telle situation pourrait entrainer un gel complet d'environnement d'exécution. Aussi, une accumulation de la mémoire due à une suite d'appels récursifs pour la mise en place de l'adaptation pourrait remplir complètement la mémoire de la machine.

- La **couverture de l'espace d'adaptation** d'un composant est un indicateur de la proportion d'états supportés par un composant adaptatif. Ce dernier peut avoir des candidats de substitution et la couverture peut être calculée pour n'importe quel composant d'une hiérarchie de composition. Une couverture inférieure à 100% indique que certains états que le système peut atteindre ne sont pas couverts par le composant analysé. L'atteinte de tels états entraine un comportement indéfini. Ainsi, une application pourrait terminer son exécution avec une erreur ou produire de mauvais résultats de façon silencieuse (si l'état non couvert n'est pas détecté).

5.1.6 Commentaires

Dans les sections précédentes, les propriétés de diverses métriques ont été présentées. Plus particulièrement, dans la section 5.1.1, il a été noté que les métriques de la littérature actuelle sont souvent basées sur un jugement subjectif de caractéristiques des composants d'un logiciel et souvent utilisées pour comparer des systèmes en termes de niveau d'adaptativité. Dans les sections suivantes, des métriques dont les calculs sont basés sur la structure induite par les patrons de conception du présent projet de recherche sont proposées comme des solutions pouvant aider à quantifier la qualité de façon pleinement objective. Tel que résumé à la section 5.1.5, les métriques proposées approchent des aspects différents de la qualité, mais conservent le but de rester dans un domaine objectif.

Dans les trois métriques proposées, il a été décidé de dédier un article complet à une métrique afin de permettre une description la plus complète possible. Le choix s'est arrêté sur la métrique *couverture de l'espace d'adaptation*, car elle est une métrique difficile à calculer manuellement et constitue un indicateur de problèmes critiques. Un problème critique a comme conséquence l'arrêt du système ou la violation de ses spécifications. Le défi entourant son calcul est au niveau de la spécification des espaces d'états : la couverture doit pouvoir être calculée depuis une fonction générique qui supporte plusieurs types de domaines (énuméré et discret). L'explosion combinatoire qui existe lorsqu'un contexte est modélisé par la combinaison des états possibles des différents paramètres doit être abordée et possiblement résolue par l'utilisation d'une interface commune pour ces domaines.

En ordre, il serait nécessaire, dans le futur, de se pencher sur la métrique de *stabilité de l'adaptation* puisque l'existence d'instabilité peut également induire l'arrêt du système. Cependant, l'effort d'implémentation apparaît inférieur, car il ne nécessite pas de nouveaux développements au niveau des artéfacts présents dans les patrons de conception (en opposition aux nouveaux types d'espaces d'états implémentés pour la métrique de *couverture de l'espace d'adaptation*). Ensuite, la *profondeur de l'arbre de substitution* est une mesure de complexité et de performance. Son impact est un peu plus superficiel puisqu'il se trouve au niveau de la qualité à l'utilisation du logiciel, soit la qualité perçue par l'utilisateur. Néanmoins, une trop grande complexité peut induire un arrêt apparent du système. Dans ce cas, le système pourrait ne plus répondre pendant un long moment du point de vue de l'utilisateur.

5.2 Abstraction des espaces d'états

Dans le premier article, les espaces d'états utilisés pour définir les espaces d'adaptation des composants et le domaine de valeurs possibles pour les paramètres et moniteurs étaient basés sur des ensembles (*sets*). Cette structure de données définit un ensemble de valeurs non ordonné. Cependant, cette structure n'est pas pratique pour la définition de larges domaines de valeurs, par exemple les valeurs entières entre 1 et 10000. En effet, il serait nécessaire de mettre en mémoire toutes les valeurs alors qu'elles suivent une règle et pourraient être générées (ex. : valeurs de 1 à 10000 avec un intervalle de 1). De plus, les opérations sur des ensembles comportant un nombre élevé d'éléments sont peu performantes.

Pour mieux travailler avec les espaces d'états, une abstraction a été créée. Les implémentations de différents types d'espaces d'états permettent la déclaration d'espaces énumérés (basé sur les ensembles, tout comme pour le premier article) et d'espaces discrets. Cette dernière implémentation permet de spécifier un nombre de début et de fin ainsi qu'un écart et d'obtenir un domaine de valeurs comme l'exemple précédent. Les opérations effectuées sur cette structure de données comme l'union et l'intersection ne nécessitent pas d'itérer à travers toutes les valeurs possibles comme pour les ensembles, et la mémoire est limitée à trois nombres. Un gain en performance est donc attendu pour de grands ensembles, mais le principal avantage est surtout que l'abstraction des espaces d'état offre une interface générique pour leur manipulation dans la librairie AdaptivePy. Les parties de la librairie qui tenaient pour acquise l'utilisation d'ensemble ont été révisées pour utiliser les opérations génériques.

Le développement d'une abstraction pour les espaces d'états est également dû au fait que les calculs au niveau de la métrique de couverture des espaces d'adaptation doivent pouvoir être faits pour tout type d'espaces d'états. Ainsi, en utilisant des opérations abstraites fournies définies dans l'interface de l'espace d'états, une seule méthode de calcul est nécessaire pour tout composant adaptatif. L'implémentation du calcul, dérivée des équations présentées au tableau 6.2, peut donc prendre en argument n'importe quel composant adaptatif défini par AdaptivePy. En plus, pour des implémentations d'espace d'états supplémentaires, aucune modification à l'implémentation des métriques n'est requise, tant et aussi longtemps qu'elles supportent les opérations de l'interface commune.

5.3 Détails d'implémentation de la couverture de l'espace d'adaptation

Comme le deuxième article couvre uniquement la définition formalisée et l'utilisation des métriques de couverture de l'espace d'adaptation, la présente section vise à présenter des informations additionnelles concernant leur implémentation. Pour des détails plus précis sur les calculs eux-mêmes, se référer à la section 6.3.

5.3.1 Implémentation de la couverture récurrente

Dans l'article, l'implémentation de référence de la couverture récurrente (voir tableau 6.1, ligne « RSC ») n'est pas discutée en profondeur. L'implémentation actuelle est considérée comme non optimale, ainsi il est opportun de présenter la solution actuelle et les pistes pour proposer une approche plus performante.

La stratégie actuellement utilisée pour effectuer le calcul est d'itérer dans tous les états définis au moins une fois. Pour ce faire, un espace d'états est généré comme étant l'union des espaces d'adaptation des candidats de substitution. La fonction itère dans les états de ce dernier et, pour chaque état, vérifie le nombre de composants qui le définissent. Si le nombre de composants le définissant est supérieur à 1, le nombre de définitions récurrentes (le nombre total de définitions, moins un) est ajouté à un compteur. Une fois tous les états visités, la couverture récurrente est obtenue en effectuant le ratio entre le compteur et le nombre d'états possibles.

On suppose que cette approche n'est pas optimale puisqu'il est nécessaire d'effectuer la visite par force brute de tous les états définis. Comme une implémentation abstraite des espaces d'états a été faite, il est attendu que des opérations sur ces structures de données puissent fournir une réponse sans nécessiter de visiter chaque état explicitement. Par exemple, une solution pour deux composants serait d'abord d'effectuer l'intersection de leur espace d'états respectif. Ensuite, il suffirait d'effectuer le produit des ratios correspondants au nombre d'états résultant de l'intersection par rapport au nombre d'états possibles de chacun des paramètres définis. Cependant, les calculs pour un nombre n de composants ne sont pas aussi simples. Comme la performance du calcul de la métrique n'a pas été problématique et que le projet de recherche actuel n'avait pas comme but l'optimisation du calcul des métriques, l'exploration d'une méthode de calcul plus efficace a été réalisée, mais n'a pas donné les résultats escomptés. Il advient donc de travaux futurs à optimiser

le calcul de la couverture récurrente qui est le goulot d'étranglement de l'implémentation actuelle.

5.3.2 Pseudo-code de l'implémentation

Un pseudo-code permet de démontrer la façon dont un algorithme est implémenté sans être lié à un langage en particulier. Le pseudo-code de l'implémentation des métriques est donné à l'algorithme 1. L'algorithme 2 présente l'obtention d'un espace d'adaptation avec des moniteurs à la place des paramètres. L'algorithme 3 présente le calcul de la couverture d'un espace d'adaptation.

Il est à noter que lorsque l'algorithme 1 est utilisé pour effectuer un calcul de couverture effective (le drapeau `effective` est à *True*), le même algorithme de calcul (algorithme 3) est utilisé. Cependant, les paramètres, qui constituent les clés dans *adaptationSpace*, sont échangés dans l'algorithme 2 pour les moniteurs correspondants, de sorte que le calcul de couverture reste le même. Dans l'algorithme 3, la ligne 4 est générique et fonctionne dans les deux cas, dans la mesure où la classe moniteur est dérivée de la classe paramètre[1].

[1]Alternativement, pour un langage supportant le *duck typing* comme Python, la méthode `possible_values` définie par les artéfacts Parameter et Monitor du patron Moniteur (voir figure 4.1) ne nécessite pas de relation d'héritage explicite. Le *duck typing* permet d'utiliser deux classes partageant des définitions de méthodes de la même façon sans nécessiter un lien d'héritage. Dans le cas présent, comme la seule méthode utilisée est `possible_values` et que les deux classes définissent une méthode de ce nom, les deux classes peuvent être utilisées dans le contexte du calcul de la métrique.

Algorithme 1 Couverture modélisée et effective de l'espace d'adaptation

```
1: function CompositeASCoverage(component, effective, asPrimitive)
2:     coverage ← 0.0
3:     if not asPrimitive and component has substitution candidates then
4:         if effective then
5:             candidates ← instances of candidates of component
6:         else
7:             candidates ← classes of candidates of component
8:         recurrentCoverage ← recurrent coverage of candidates
9:         sumCoverage ← 0.0
10:        for all candidate in candidates do
11:            sumCoverage ← sumCoverage + ASCoverage(candidate, effective, True)
12:        coverage ← sumCoverage − recurrentCoverage
13:    else if component has adaptation space then
14:        space ← adaptation space of component
15:        if effective then
16:            space ← monitor space corresponding to parameters in space   ▷ See algo 2
17:        coverage ← adaptation space coverage of space                    ▷ See algo 3
       return coverage
```

Algorithme 2 Espace d'adaptation avec moniteurs

```
1: function GetMonitorSpace(adaptationSpace, parameterValueProvider)
2:     monitorSpace ← map of (monitor, state space)
3:     for all parameter, monitor provided by parameterValueProvider do
4:         space ← adaptationSpace[parameter]
5:         if adaptationSpace has parameter as key then
6:             monitorSpace[monitor] ← adaptationSpace[parameter]
       return monitorSpace
```

Algorithme 3 Calcul de la couverture d'espace d'adaptation pour un composant primitif

```
1: function PrimitiveASCoverage(adaptationSpace)
2:     coverage ← 1.0
3:     for all parameter, space do
4:         possibleValues ← parameter's possible values
5:         definedStates ← intersection between possibleValues and space
6:         numerator ← size of definedStates
7:         denominator ← size of possibleValues
8:         coverage ← coverage ∗ (numerator/denominator)
       return coverage
```

5.4 Conclusion

Dans ce chapitre, différentes métriques potentielles ont été présentées qui avaient pour but de participer à la V&V d'un logiciel dont les éléments adaptatifs ont été implémentés avec les patrons présentés à la section 4.3. La réalisation que la structure mise en place par l'utilisation des patrons de conception ne garantit pas une implémentation sans faille indique que le développement de nouvelles méthodes de V&V est nécessaire.

La métrique choisie pour le présent projet de recherche est la couverture de l'espace d'adaptation. Cette métrique apporte une aide importante pour minimiser les situations de comportements indéfinis. Comme les patrons de conception se concentrent uniquement sur la structure pour l'adaptation, l'utilisation d'une telle métrique offre une façon d'augmenter la confiance des développeurs en la réalisation d'une application qui utilise cette structure. Comme le chapitre 6 focalise sur la formalisation et l'utilisation concrète des deux versions de la métrique, plusieurs détails d'implémentation sont omis. Dans le présent chapitre, le pseudo-code de la fonction implémentant la métrique est donné à la section 5.3 ainsi que des détails sur les stratégies de calculs utilisés. Une critique de l'implémentation est également présentée, particulièrement au niveau de l'implémentation de la couverture récurrente qui est fonctionnelle, mais considérée non optimale.

CHAPITRE 6

Article 2 : Metrics for adaptation space coverage evaluation in adaptive software

Avant-propos

Auteurs et affiliation :

S. Longchamps : étudiant à la maîtrise, Université de Sherbrooke, Faculté de génie, Département de génie électrique et de génie informatique.

R. Gonzalez-Rubio : professeur, Université de Sherbrooke, Faculté de génie, Département de génie électrique et de génie informatique.

Date de soumission : 13 mars 2017

Revue : Proceedings of the Tenth International C* Conference on Computer Science & Software Engineering

Titre français : Métriques pour l'évaluation de la couverture de l'espace d'adaptation dans les logiciels adaptatifs

Contribution au document : Les métriques présentées sont issues de l'une des métriques proposées à l'étape de recherche (voir section 5.1). Elles sont des spécialisations de la métrique de couverture de l'espace d'adaptation présentée à la section 5.1.4. Les métriques présentées servent à vérifier que l'application utilisant la structure induite par l'utilisation des patrons de conception supporte bien les situations atteignables par le système. Une présentation formalisée et leur utilisation dans une situation concrète permettent une excellente compréhension de la métrique et supportent les sous-objectifs de l'objectif secondaire présentés à la section 1.3.

Résumé français : Les logiciels adaptatifs sont typiquement complexes et l'assurance qualité interne est difficile à effectuer. Les métriques constituent des outils pour évaluer la qualité logicielle de façon quantitative et pour aider les développeurs dans l'identification de régions qui nécessitent des améliorations par rapport à diverses préoccupations. Une préoccupation spécifique aux logiciels adaptatifs est le support par les composants du nombre élevé d'états contextuels qu'un système peut atteindre. Lorsque modélisé sous

forme d'espace d'états, cet ensemble d'états contextuels est appelé un espace contextuel. Des composants adaptatifs démontrent leur support pour une région de l'espace contextuel d'une application par le biais d'un espace d'adaptation. Le présent article propose deux métriques afin d'aider à cette tâche : la couverture modélisée de l'espace d'adaptation et la couverture effective de l'espace d'adaptation. Ces métriques sont utilisées, respectivement, à la conception et à l'exécution. Elles offrent une valeur entre 0.0 et 1.0, indiquant la proportion des états supportés par rapport à ceux atteignables par le système. Les métriques sont présentées dans un format inspiré de l'ISO/IEC 25023. Elles sont utilisées pour analyser une application typique dans le cadre d'une étude de cas et une interprétation des résultats est offerte. Trois types d'actions correctives sont présentées et leur impact sur la valeur des métriques est expliqué. Une implémentation du calcul des métriques fut développée dans le langage Python et rendue disponible librement. Dans cet article, nous montrons comment les métriques peuvent être utilisées pour identifier les problèmes et mettre en place des actions pour les corriger afin d'obtenir une implémentation de l'adaptation de qualité. L'étude de cas et les actions correctives présentées mettent en lumière les gains offerts par l'utilisation des métriques. Sans les métriques, les problèmes présentés peuvent être négligés et amener à des comportements indéfinis dans le logiciel adaptatif.

6.1 Introduction

Adaptive software is typically complex and internal quality assurance is difficult to achieve. While diverse work exists on qualitative and quantitative means of analyzing the quality of adaptive software [38], the structural aspect of software implementation is rarely discussed. Many researchers have proposed metrics to compare adaptive systems and to provide hints to developers regarding which implementation is "better" than the other with regards to various concerns. Another proposed solution is the evaluation of the degree of adaptability of an application such that quantitative requirements can be expressed and checked [37]. The current work takes a different direction by assisting developers in the evaluation of structural support of contexts an adaptive application can encounter.

Adaptive software is different from traditional software in that their architecture can be dynamic and large arrays of contexts lead to large arrays of possible behaviors. To support all of these contexts, developers can create components and modules which take into account the variability provided by the contextual information. For example, a system can be required to exhibit a different behavior depending on the time of the day. A developer can design a component which will behave in some way, valid for 9AM to 5PM, and another which will behave in another way, valid from 6PM to 9AM. From this situation, we can instinctively see that the hour between 5PM and 6PM is not supported by any component. The identification of more complex contexts with many more variables and/or components would lead to even more complex unsupported cases. There is a need to help developers verify that their design supports the ranges of contexts the system can encounter.

Our proposal is to devise metrics to express the proportion of contexts covered by an adaptive component. Because components can be composed into more complex components, components at any level of a components hierarchy can be analyzed. The idea is to assist developers in verifying that components have adequate coverage of contexts such that an application will not exhibit undefined behavior for contexts part of its specifications. We expect this assistance to be especially useful for complex adaptive components architectures where the number of contexts is beyond computational capacity. It must be stressed that the metrics we proposed do not aim at verifying that adaptation mechanisms and adaptive behavior of an application fulfill adaptation requirements. In fact, we assume that these concerns are verified separately by specialized means, such as dedicated tests. Using this assumption, we aim to verify instead if components are assembled in a hierarchy which supports specified contexts. We also use specifications introduced in our previous work [27] on the way adaptive components are expected to be composed to efficiently compute the metrics.

The remainder of this paper is organized as follows. Related work is presented in Section 6.2. The proposed metrics are presented in Section 6.3. A case study with a sample program is presented in Section 6.4. Results and analysis of the sample program with some actions to help developers are presented in Section 6.5. The usage and limitations of the metrics are discussed in Section 6.6. Finally, a conclusion with future work is presented in Section 6.7.

6.2 Related work

This section presents related work, starting with previous research efforts. Then, other work related to adaptive software verification is presented and compared to the current work.

6.2.1 Previous research work

Previous work [27] focused on identifying and unifying common solutions to express and use adaptation contexts in a scalable manner. For example, we used state spaces to express domains of contexts (a "context space") adaptive software can encounter. In a context space, each contextual variable is a dimension and each contextual variable define a set of possible values. A context is a combination of one value per dimension which represents the contextual state of the system. For example, for contextual variables A and B with possible values A = {1, 2} and B = {2, 3}, the context space would be {(1, 2), (1, 3), (2, 2), (2, 3)}. The idea was to use state spaces to express information about contexts for different concerns of adaptive software. To do so, some artifacts which use state spaces were proposed :

- **Parameter :** Represents a contextual variable as modeled at design time. Its state space is used to define all the possible values of a contextual variable.
- **Monitor :** Represents a means of obtaining values for a parameter. It provides a subset of values defined by a parameter. This subset is also defined as a state space.
- **Adaptive component :** Represents a component which exhibits adaptation. It adapts itself to context variations detected when monitors provide different values for parameters it is aware of. These parameters and the domain of values the component supports are defined in a state space called "adaptation space". An adaptive component can be composite, that is, it can be substituted by other adaptive components with each their own adaptation space. The adaptation space of the composite

adaptive component is then the union of the substitution candidates' adaptation spaces.

These artifacts were formalized in design patterns and implemented in a reference library, AdaptivePy[1]. While a case was made to show how their use increases quality in adaptive software, previous work lacked metrics to quantitatively analyze their use in concrete situations. Moreover, composite adaptive components could lead to large hierarchies of adaptive components supporting a subset of the context space which could not be easily analyzed. For the current work, we identified the need to verify that adaptive components, composite or not, support a specified proportion of contexts.

Our proposal is that this verification can be realized with the help of metrics like the adaptation space coverage of each adaptive component. Concretely, the idea is to verify that components of an application support the necessary contextual states that can be reached. By composing adaptive components with different adaptation spaces, a greater coverage of reachable states is attained. This allows to concentrate quality assurance efforts on small ranges of specific contexts defined in specialized components. The alternative would be to test each component in all possible contexts using every parameter, leading to combinatorial explosion.

6.2.2 Verification and validation metrics

Developing quality software is a challenge which relies on the ability to evaluate quality and to devise ways to improve solutions to attain higher quality. Verification and validation (V&V) techniques are used to assess the quality of software applications. One type of V&V technique is the use of metrics, which are quantitative indicators of the quality of a software component with regards to a concern.

When it comes to adaptive software, additional metrics are needed. Raibulet *et al.* [38] have investigated the evaluation of "self-adaptive systems". The analyzed approaches served as inspirations for the current work, but we consider our focus is different in that it aims to provide verification of the support for a system's context space by its components. Existing literature focuses either on the comparison between adaptive systems [32, 35] or on evaluation of mechanisms [46] to attain adaptation goals. What we want to verify is independent of specific adaptation mechanisms. Also, instead of evaluating the degree at which a system is adaptable or adaptive, we wish to quantify the degree of support for a system's context space components offer.

[1] `https://gitlab.com/memophysic/adaptive_py_lib`

Kaddoum *et al.* [22] proposed metrics for the evaluation of adaptive systems with the motive to be able to better compare applications in terms of adaptiveness. Metrics are divided in three categories : "methodological", "intrinsic" and "runtime evaluation" criteria. The first two criteria are closest to our goal, but they are limited to the evaluation at a component level (count number of components) and therefore do not quantify adaptation within components. The biggest downside is that they rely heavily on qualitative inspection of components to judge if they are considered adaptive. Nevertheless, interesting metrics are proposed and seem compatible with the conceptual framework defined in our previous work.

Perez-Palacin *et al.* [37], like Kaddoum *et al.* [22], proposed metrics for the evaluation of adaptive systems, but with the focus on the relationship with quality of service. The adaptability of a system is presented as the ability to reconfigure connections between components of compatible provider/consumer service. The granularity level is at the service level and the goal is to quantify how adaptable are specific services or on average in the system. Our previous work proposed that components offering the same service can be "substitution candidates" for a composite adaptive component. Perez-Palacin *et al.* [37] propose the "absolute adaptability of a service" as the number of substitution candidates for a service. This design-time metric is used to express the degree of adaptability of a service. At runtime, they propose the "relative adaptability of a service" as the ratio of substitution candidates concretely used over the number of existing substitution candidates. The differentiation between design-time and runtime leads us to propose a version of the adaptation coverage metric for each phase. The main downside of the work presented is the granularity level since the ability for the substitution candidates to provide the service is assumed, regardless of the contextual state the system is in.

Another interesting research not mentioned in [38] is the work of Tamura *et al.* [44] with regards to runtime verification of self-adaptive systems. Their goal was to look at the challenges of providing certifiable V&V techniques aimed specifically at adaptive systems. An important concept in their work is the identification and modeling of a "viability zone" during the lifecycle of an application, but particularly at runtime. They define a viability zone "as the set of possible system states in which the system operation is not compromised" [44]. They show how variations in this variability zone at runtime needs to be monitored and corrective actions taken to ensure the system stays within it. In our conceptual framework, a component's adaptation space represents the viability zone within the application's context space. So far, our work has looked solely at fixed adaptation spaces, but our conceptual framework does not forbid them to change at runtime. The

most interesting link we can do with our work is that the runtime version of our metrics could be used at runtime to detect if the viability zone of a component is incompatible with the context space of the system. While many concepts are analyzed in their work, no V&V metrics are proposed. We expect that our work will help alleviate some of the mentioned challenges, for example by reducing the verification space which is subject to combinatorial explosion when all parameters are considered during the evaluation of a complete system.

Raibulet *et al.* [38] offer some interesting insights to the problems the work they analyzed presented. For example, they mention how important a case study and tool support are to the application of the proposed methods. In fact, many papers did not provide tools and limited their case study to the application of their approach without further information on how to interpret the results. They also mention that no work includes a complete set of evaluation mechanisms. It is then indicated that new mechanisms and refinements of previous ones are needed.

The current work aims at providing all these important elements such that the metrics can be widely understood and used for V&V of various applications. An implementation of the metrics is freely available as open source software [2] and Section 6.5 provides a full picture of how the results can be interpreted and corrective actions taken.

6.3 Metrics

This section introduces two new metrics based on state space coverage for adaptive software : ***modeled* and *effective* adaptation space coverage**. Both are measures of state space coverage, but are based on different sources of reachable values. These metrics are described in a format inspired by ISO/IEC 25023. This standard is part of the Systems and software Quality Requirements and Evaluation (SQuaRE) series of International Standards. As shown in Fig. 6.1, various elements are defined in the context of SQuaRE, with ISO/IEC 25023 focusing on "provid[ing] measures including associated measurement functions for the quality characteristics in the product quality model" [21]. To provide a complete picture of the scope of the proposed metrics, each element of the SQuaRE series is addressed directly. While modeled and effective adaptation coverage are different metrics, they are similar and therefore will be presented conjointly.

[2] `https://gitlab.com/memophysic/adaptive_py_metrics`

Tableau 6.1 Adaptation state space measures

Name	Description	Measurement functions and QMEs
State space coverage (SC)	What proportion of the state space is valid ?	X = A/B A = number of valid states B = number of possible states
Recurrent state space coverage (RSC)	What proportion of a state space is recurrently defined by a set of state spaces ?	$X = (\sum max[0, A_i - 1]) / B$ A_i = number of times the i-th state is defined B = number of possible states
NOTE : The notation $max[a, b]$ indicates the maximum value between a and b.		
Primitive adaptation space coverage (PASC)	What proportion of contextual state space is supported by a primitive component ?	X = A/B A = number of states defined in a primitive component's adaptation space B = number of possible states the component is aware of
Composite adaptation space coverage (CASC)	What proportion of contextual state space is supported by a composite component ?	$X = (\sum A_i) - B$ A_i = substitution candidates' adaptation space coverage, that is the measured value of $PASC$ for the i-th substitution candidate B = substitution candidates' recurrent adaptation space coverage, that is the measured value of RSC for the substitution candidates

Because the focus of the current work is also on quality measurements, table 6.1 was produced with the goal of presenting generic metrics. The base metric is the state space coverage, which is adequate for computing the coverage over the complete system. However, there exists an assumption for the adaptation space defined by adaptive components : any parameter not explicitly defined is considered entirely supported. In this case, the component is said "agnostic" to the parameter. From this, the primitive adaptation space coverage uses the number of possible states that a component is *aware of*, that is only those originating from parameters explicitly defined in its adaptation space. For adaptive components with substitution candidates ("composite" components), instead of computing the adaptation space by unionizing those of the candidates, we make use of the PASC to compute their respective coverage. That way, we can obtain the composite adaptation space coverage (CASC) simply by summing up the PASC of the candidates and subtracting the coverage related to the states defined multiple times by each candidate. This last metric is formalized as the recurrent state space coverage, applicable to a set of state spaces. With this scheme, it is possible to compute the coverage of any component within

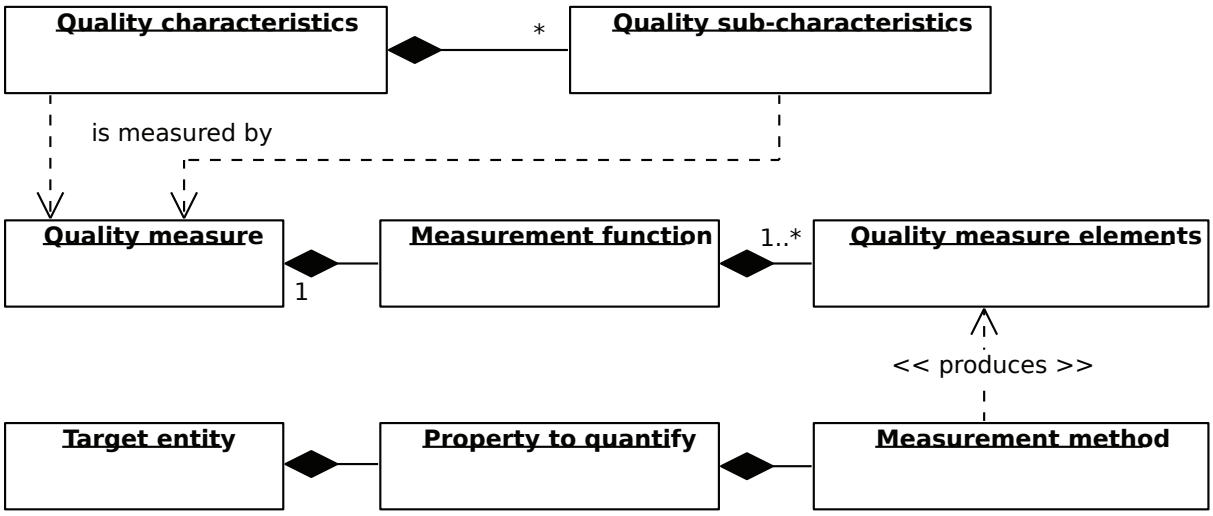

Figure 6.1 Relationship among elements defined by the SQuaRE series of International Standards [21]

a composition hierarchy by requesting the primitive coverage of its substitution candidates. This is possible because all adaptive components are required to provide the adaptation space they support explicitly. Therefore, all substitution candidates are considered black boxes and each composition layer can be analyzed independently from each other.

In the remainder of this section, the elements defined in the SQuaRE series are addressed with a description for each of the metrics' context and properties.

Quality characteristic The quality characteristic being measured is the structural support for adaptation contexts of an application. We seek to determine if an application has the capability to support ranges of contexts it is expected to encounter.

Quality sub-characteristic When verifying component-based software, the structural support of contexts of an application relies on the support provided by each of its components. In adaptive software, the structure of components can change depending on the context. A quality sub-characteristic is therefore the structural support for adaptation contexts offered by components in a dynamic environment.

Measure definition The structural support of contexts will be measured using *adaptation space coverage*, that is, state space coverage of components' adaptation space. It is defined as the proportion of the number of states supported by a component as defined in its adaptation space compared to the number of reachable states, either defined by

parameters as a *modeled space* or by monitors as an *effective space*. This last distinction is reflected in variable B for the state space coverage function (see table 6.1) : the "number of possible states" can be those provided by parameters or by their bound monitors. From this fact, the proposed coverage metrics are labeled "modeled coverage" when based on parameters and as "effective coverage" when based on monitors.

Measurement functions The functions for modeled adaptation space coverage (MASC) and effective adaptation space coverage (EASC) are given in table 6.2. A few definitions of terms are needed to understand the functions.

An adaptive component c has an adaptation space σ_c defined as a subspace of a context space where every contextual variable is a dimension. The states of each contextual variables are defined at design-time using parameters and at runtime by monitors. Therefore, two types of context spaces respectively exist : modeled context space Ω and effective context space ω. A composite component has n substitution candidates s with each an adaptation space σ_{s_i}. When looking at a context space constrained to the parameters known by a set of components $c_{0...n}$, the modeled context space is $\Omega_{c_{0...n}}$ and the effective context space $\omega_{c_{0...n}}$.

The terms found in table 6.2 are summarized below.

- σ_{s_i} : Adaptation space of substitution candidate s_i for a given component
- $\Omega_{s_i}/\omega_{s_i}$: modeled/effective context space of substitute candidates for a given component
- $\Omega_{s_{0...n}}/\omega_{s_{0...n}}$: modeled/effective context space of all substitute candidates for a given component
- $c(x)$: Number of states in the state space x (where x could be the state space corresponding to σ, Ω or ω)
- $r(s_{0...n})$: Number of recurrent states for all substitute candidates (obtained with an algorithm)

Tableau 6.2 Measurement functions for metrics

Metric	Function	
MASC	$CASC($	$A_i = PASC(A = c(\sigma_{s_i}), B = c(\Omega_{s_i})),$ $B = RSC\,(A_j = r(\sigma_{s_{0...n}}), B = c(\Omega_{s_{0...n}})))$
EASC	$CASC($	$A_i = PASC(A = c(\sigma_{s_i}), B = c(\omega_{s_i})),$ $B = RSC\,(A_j = r(\sigma_{s_{0...n}}), B = c(\omega_{s_{0...n}})))$

We can see that the functions from table 6.2 are implemented using the generic measurement functions presented in table 6.1. More specifically, both MASC and EASC metrics use the $CASC$ measurement function. Note that when using this measure on a *primitive* component instead, the equation is equivalent to $PASC$ since $RSC = 0$ and there are no substitution candidates, therefore the only primitive for the sum of $PASC$ is itself. Resulting values range from 0.0 to 1.0, where 1.0 is the ideal value. Values < 1.0 indicate that there are unsupported contextual states by the component.

Quality measure elements (QMEs) QMEs are the *number of states* defined in state spaces of different artifacts.

- **Number of modeled context states** as defined by parameters. They represent the reachable contexts as modeled at design time. Each parameter provides a dimension of the state space.
- **Number of effective context states** as defined by monitors. They represent the reachable contexts which can be encountered at runtime for a given set of monitors. Like parameters, each monitor provides a dimension of the state space.
- **Number of component adaptation states** as defined by an adaptive component. They represent the viability zone of the component, that is, the contexts it supports. The states are a subset of those defined by the parameters.

Measurement method The method used is the same for all QMEs : acquisition and inspection of state spaces from artifacts (parameters, monitors and adaptive components). At design time, the state space of the parameters the component is aware of can be statically acquired, rendering static analysis of modeled adaptation space coverage possible. At runtime, the state space of the monitors can be acquired by obtaining the list of monitors corresponding to the list of parameters the adaptive component is aware of (e.g. from a monitoring manager entity).

Property to quantify The property to quantify is the adaptation space coverage of an adaptive component. This quality property indicates what proportion of contexts that an application can encounter a certain component explicitly supports.

Target entities Adaptive components, parameters and monitors.

6.4 Development

The computation of the metrics was implemented to demonstrate its application. Because the necessary artifacts were implemented as part of our previous work in the library AdaptivePy, it was reused and improved to support more state space operations necessary for the calculations. Originally, the library only supported enumerated state spaces. The new version adds the abstract handling of state spaces along with basic operations such as unions and intersections.

Using discrete state spaces, it is possible to efficiently define parameters which have a very large state space. One example is how temperature is represented for measurements ranging from -40° to 60° Celcius at the precision of 0.5°, expressed as [-40, 60, 0.5]. This leads to a state space of $\frac{60-(-40)}{0.5} = 200$ states. Additionally, a second parameter could be used to reinterpret these values in more human-readable terms such as {cold, normal, hot} by using thresholds within a monitor. Since monitors can define values subset of their parameter, precision could be lower (e.g., 1°) and limited to a smaller range (e.g., -30°to 30°).

The metrics' computation is done through a single function which takes as input an adaptive component and returns a value in the domain [0, 1], representing a percentage between 0 and 100%. The proposed solution in our previous work to simplify construction of complex adaptive component hierarchies was to assume complete support of state spaces defined by substitution candidates. This means that there is no need to query further than the direct substitution candidates to obtain a complete definition of the adaptive component's adaptation space. This assumption was used to automatically compute the default adaptation space of a composite adaptive component as the union of its substitution candidates'.

For the metrics' computation, this means that substitution candidates are considered primitive components because they are expected to support the complete adaptation space they provide. Any undefined states can be manually supported by parametrization within the composite adaptive component or by further constraining its adaptation space. This reduces the problem of state space size explosion by only requiring the computation of recurring coverage to a single set of primitive components for any composite adaptive component.

Computation of modeled coverage is done without the need to instantiate components by using statically defined possible values for both parameters and components. For effective coverage, there is a need to instantiate components because monitors are registered to one

or many parameter value providers (PVP) and a component can be bound to only one PVP. This constraint was proposed in our previous work [27] to enable agreement with regards to context between collaborating components. An implication of this constraint is that depending on which PVP the component is bound to, different monitors can be set and cover equally different regions of the viability zone. Two sets of monitors can therefore provide different coverage values.

A sample program was written to demonstrate the use of the metrics. The program represents an adaptive caging system for a zoo which allows the zookeepers to distribute and transfer animals to appropriate cages. Two contextual variables are used to make adaptation decisions and are modeled as parameters :

- **AnimalSize (AS)** : {small, medium, large, gigantic}
- **Temperature (T)** : [-40, 60, 0.5]

An **AnimalContainer** interfaces describes what is expected to be accomplished by the adaptive component : an animal can be set in the container, it can be checked for emptiness and it can be cleaned. An adaptive component using substitution therefore needs to adapt to the various animal sizes and temperatures. Three types of animal containers are defined as three adaptive components with the following adaptation spaces :

- **Cage** : AS={small, medium}, T=[10, 60, 0.5]
- **Barn** : AS={small, medium, large}, T=[-40, 30, 0.5]
- **VirtualEcosystem (ES)** : AS={gigantic}

Note that since VirtualEcosystem doesn't define any adaptation over the temperature parameter, it is agnostic to it, meaning it supports any temperature and isn't expected to adapt to temperature changes.

6.5 Results

The sample program presented in 6.4 has been implemented using AdaptivePy. The three primitive adaptive components along with a composite adaptive component have been used as input for the metrics computation function, both as modeled and effective. Monitors were created to provide a single random value among the possible values of their parameter. The results are presented in table 6.3. From these results, we can see that each substitution candidate has partial coverage, with Barn being the most flexible since it supports more than half of the states compared to a quarter for Cage and VE. Interestingly, Composite

has a coverage of 92.5%, while the sum of its candidates yields $25 + 52.5 + 25 = 102.5\%$. This means that there is $102.5 - 92.5 = 10\%$ of recurrent coverage. This recurrent coverage is due to cumulative overlap of substitution candidates' adaptation space. This property can be visualized by using parameters as dimensions on a graph. Because we have only two parameters, we obtain a 2D graph shown in Fig. 6.2. We see from this figure that Barn and VE overlap from 10° to 30° and {small, medium} animal sizes. It can also be seen that the remaining 7.5% percent to be defined in Composite to attain 100% is temperatures between 30° and 60° for {large} animal size.

Tableau 6.3 Coverage of sample program with monitors over full domain

	Cage	Barn	VE	Composite
Modeled	25%	52.5%	25%	92.5%
Effective	25%	52.5%	25%	92.5%

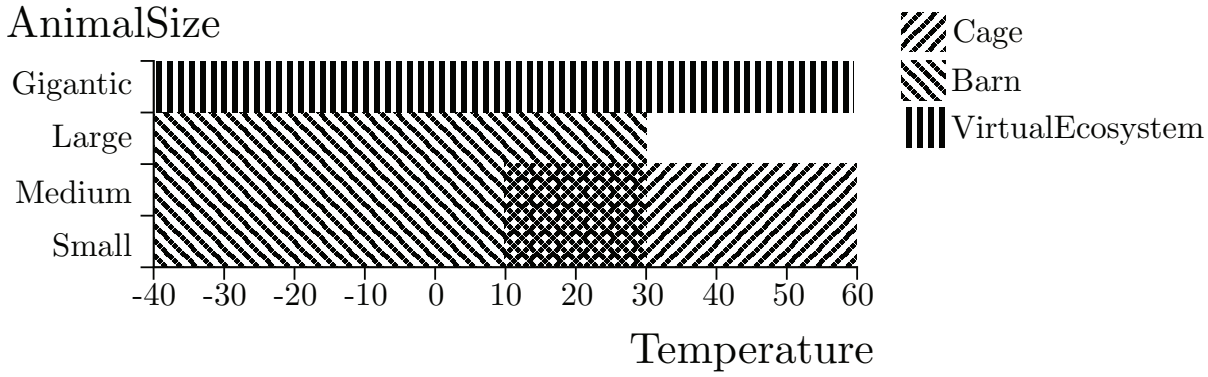

Figure 6.2 Coverage of substitution candidates for sample program with monitors over full domain

The results of the analysis of the sample program provide information regarding the correctness of the composite adaptive component. Such analysis is also a way for developers to evaluate the validity of their implementation. Table 6.3 shows that there is a need for action to be taken in order to make the composite component comply with the assumption of supporting the full adaptation space of its parts. Some actions are discussed in the following.

Add substitution candidate The undefined part could be supported by an additional substitution candidate, for example a "HeatedArea" which would support AS={large, gigantic}, T=[10, 60, 0.5]. This would lead to the results in table 6.4. We see that the coverage for the composite reaches 100%, while the additional HeatedArea adaptive component has 20% coverage. Modeled and effective coverage are still equal since monitors'

possible values have not been changed. Figure 6.3 shows how the HeatedArea fills the gap and leads to a complete support of the context space.

Tableau 6.4 Coverage of sample program with HeatedArea substitution candidate

	Cage	Barn	VE	HeatedArea	Composite
Modeled	25%	52.5%	25%	20%	100%
Effective	25%	52.5%	25%	20%	100%

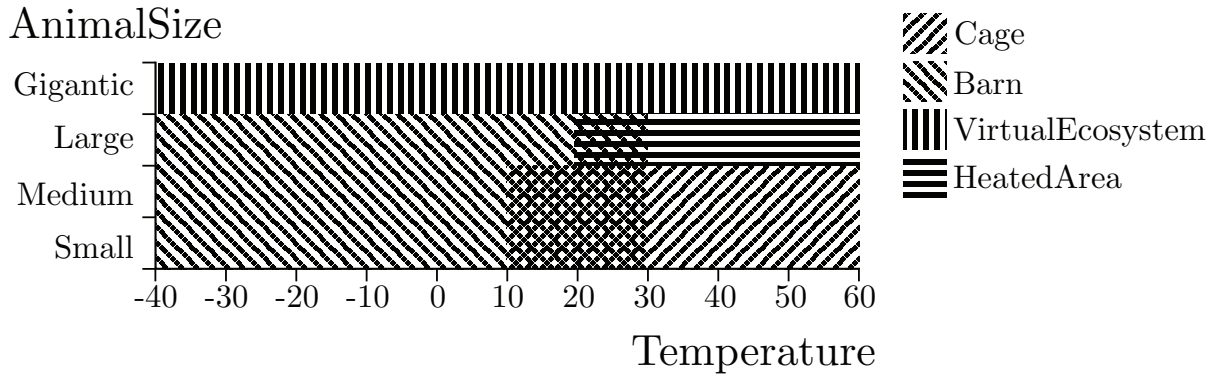

Figure 6.3 Coverage of substitution candidates for sample program with HeatedArea substitution candidate

Reduce the effective context space The undefined part could be made unreachable at runtime by reducing the monitors' possible values to a smaller viability zone. In this case, the temperature monitor could be limited to a maximum of 30°. An alternate solution is to limit the animal sizes to {small, medium, gigantic}. Using the first solution leads to the results in table 6.5. We see that effective coverage has changed in different ways for each component. Specifically, Cage and VE have decreased in effective coverage since parts of the state they defined overlapped with the constrained states. On the other hand, Barn's effective coverage has increased because the reachable state has been decreased and the monitor's constrained states were not part of the Barn's adaptation space. Note that the modeled coverage remains the same, meaning that there are still undefined states by the composite, but that they won't be reached at runtime when using the currently provided monitors. Figure 6.4 shows the reduced context space where the semi-transparent region on the right illustrates the remaining modeled space. We see that the reduced context space is fully covered without modification to the components when interpreted for runtime.

Extend a substitution candidate's adaptation space Support for the undefined part could be added to a substitution candidate by subclassing one of them and redefining

Tableau 6.5 Coverage of sample program with monitors over constrained domain

	Cage	Barn	VE	Composite
Modeled	25%	52.5%	25%	92.5%
Effective	14.28%	75%	14.28%	100%

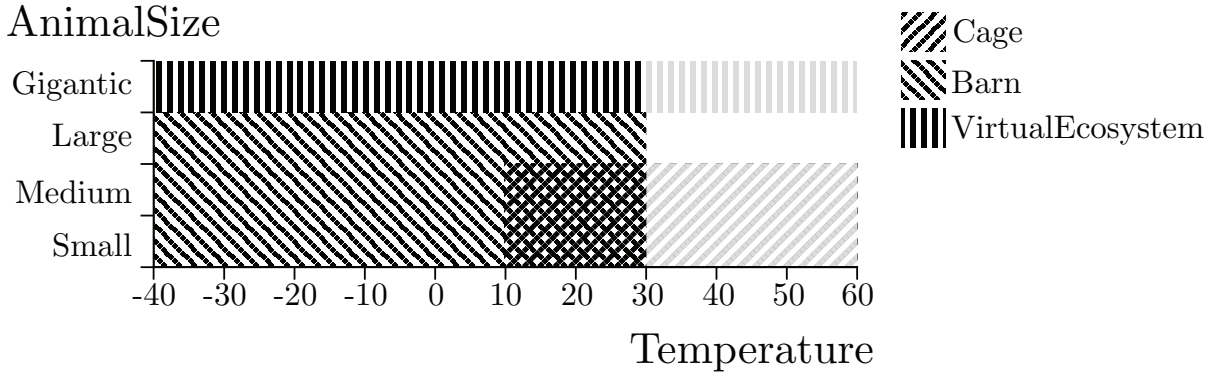

Figure 6.4 Coverage of substitution candidates for sample program with monitors over constrained domain

its adaptation space. For example, a "JumboCage" could be created as a subclass of Cage and support the large animal size. Results are shown in table 6.6. We see that the JumboCage now covers 37.5% instead of 25% and that the composite attains 100% coverage. Figure 6.5 shows how the entire modeled state space is covered by the extension of the cage component's adaptation space.

Tableau 6.6 Coverage of sample program with monitors over constrained domain

	JumboCage	Barn	VE	Composite
Modeled	37.5%	52.5%	25%	100%
Effective	37.5%	52.5%	25%	100%

6.6 Discussion

The results presented in Section 6.5 show a simple case of how the metrics can be used to verify and improve an implementation. The ideal value is 100% coverage, which can be obtained by applying one or many of the proposed corrective actions. Because there exist multiple corrective actions which can be taken to obtain larger coverage, developers are free to tune their implementation to best fit their needs.

The application of corrective actions appears to be quite obvious for the presented sample program because the parameters form a 2D space which can be displayed graphically. A

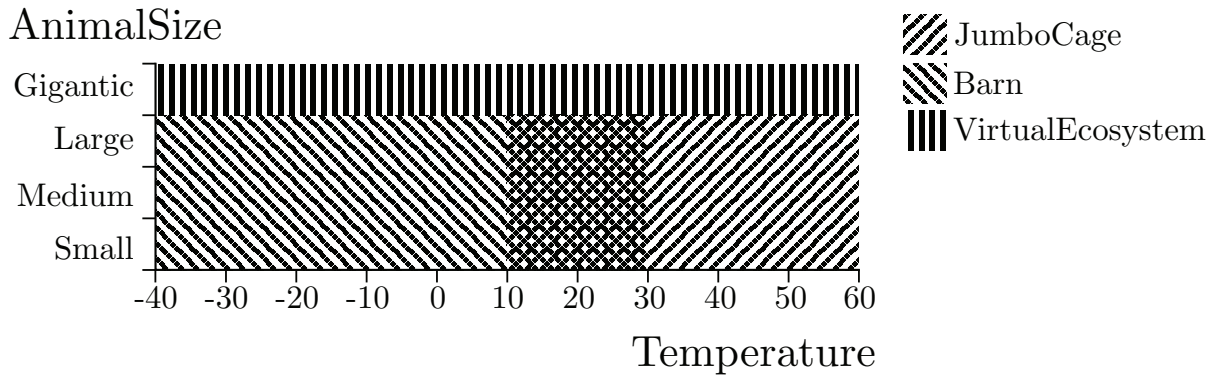

Figure 6.5 Coverage of substitution candidates for sample program with monitors over constrained domain

display limitation occurs when a component is aware of more than three parameters since they can no longer be visualized within a Euclidian space. While the actions presented can be taken with any number of parameters, identifying appropriate parameters, monitors and components to modify would remain a challenge when components are aware of a large number of parameters.

A limitation of the metrics is that it cannot be used to ensure that adaptation goals are reached. It is important to stress that the correctness of components with regards to support for their adaptation space is *assumed*. What is verified is that the adaptation space support assumption holds when components are assembled into an architecture. Also, this verification remains valid only within the specifications of the conceptual framework used in this study [27]. A main assumption brought from previous work is that substitution candidates must provide an adaptation space which is expressed as the combination of *all* the states resulting from the combinations of states derived from defined parameters. This allows any substitution candidates' coverage to be computed as primitive components and greatly simplifies the computational load for computing the coverage of a composite component in a large architecture.

An additional use of the metrics is to indicate progress of an implementation with regards to a set of supported contexts. By measuring the proportion of supported context at frequent points in time of a product's development cycle, developers can obtain indications of their progression to support all the modeled context space or the effective context space when the deployment target is known. This could be helpful to improve effort and cost previsions of context space regions support. This would be especially valid for groups of often-used parameters as experience with the implementation of their context domain could be analyzed and used for prevision.

While the implementation presented in 6.4 necessitates some artifacts to be present, the only requirement is the ability for a developer to extract state spaces related to monitoring and components/modules. Table 6.1 is a generic definition of the measures used for the metrics and can be applied with any state spaces. For very simple cases, this can also be done by hand.

6.7 Conclusion

The paper presented two metrics for the evaluation of adaptation space coverage in adaptive software : *modeled adaptation space coverage* and *effective adaptation space coverage*. Each of these metrics provide a value in the domain $[0.0, 1.0]$ which is interpreted as the percentage of the states supported by an adaptive component over the relevant context space of the system. A value of 1.0 indicates complete support by the adaptive component and prevents assures the developer that no undefined behavior is expected from a context state to be reached by the system.

A modeled coverage is based on the reachable contexts as inferred using defined parameters, independently from the deployment environment. An effective coverage is based on the reachable contexts as inferred using monitors bound to the defined parameters. Because these bindings depend on the runtime environment, effective coverage is a runtime measure as opposed to design-time for modeled coverage.

Metrics such as adaptation space coverage allow developers to verify that their system's context space will be fully supported from a high-level point of view. When developing a specific component, it allows them to quantify the proportion of the component's viability zone compared to the one that has been modeled. Unsupported states can be identified and handled, both in the design phase when the environment is modeled by parameters and at runtime when the environment is probed by monitors. This leads to a quality implementation for each layer of component substitutions.

We defined these metrics in a standardized way [21] with the inclusion of all necessary calculations and definitions. We have also shown how they are expected to be used in a case study application and how to apply various corrective actions depending on the values obtained from the metrics. Additionally, source code for the metrics' computation was made freely available. We think that these elements are sufficient to address concerns raised by [38] with regards to existing literature on the subject of quality metrics for adaptive software. We think that the metrics are easy to understand, calculate and use for adaptive software development.

Beyond this paper is the subject of tradeoff analysis which could help developers choose which corrective actions presented in Section 6.5 to apply. Such an analysis would require an understanding of non-functional requirements with regards to internal quality of the system. For example, one could prefer reducing the effective context space to achieve a less adaptive, but safer system given a certain target, while another might prefer developing an additional component to make the system as adaptive as possible. Knowing their preferences, a tool could assist them with the application of corrective actions by suggesting changes and identifying where exactly to make changes.

CHAPITRE 7

Discussion

Le présent chapitre a comme objectif d'analyser les contributions en portant un regard critique sur ces dernières. D'abord, une argumentation justifiant l'atteinte des objectifs du projet de recherche définis à la section 1.3 est présentée. Pour un résumé plus condensé du projet de recherche, consulter le chapitre 8. Ensuite, les compromis faits dans le cadre du projet de recherche sont présentés et justifiés.

7.1 Atteinte des objectifs de recherche

Pour considérer le projet de recherche comme complet, il est nécessaire de revoir les objectifs émis à la section 1.3 à la lumière des contributions et des résultats obtenus. Une argumentation au niveau de l'atteinte de l'objectif général est d'abord présentée, suivie d'une discussion sur l'atteinte des objectifs secondaires qui s'inscrivent dans l'objectif général.

7.1.1 Objectif général

Suivant l'atteinte des objectifs secondaires, il est attendu que l'objectif général qui était de trouver *quels patrons de conception permettent l'implémentation graduelle de mécanismes d'adaptation dans les interfaces graphiques* soit atteint. Par la proposition des patrons de conception visant des préoccupations précises de l'adaptation et leur utilisation dans un prototype démonstratif d'application graphique, on peut considérer que les patrons permettent au moins la mise en place d'une structure viable pour ce genre d'applications.

Dans les patrons, les mécanismes d'adaptation (l'artéfact « ChooseRouteStrategy » dans le patron Proxy routeur, voir figure 4.3.2) sont spécifiés séparément de sorte que ces derniers puissent être ajoutés graduellement et comparés dans le contexte d'une même structure. Aussi, comme il est possible de transformer un composant en version adaptative sans impacter les autres composants, le côté *graduel* de l'objectif est considéré comme atteint.

L'objectif général ne fait mention que des patrons de conception, mais il est nécessaire de constater que les patrons en soi ne peuvent assurer une structure valide pour une application donnée. Ainsi, pour compléter les patrons et réellement atteindre l'objectif général,

des métriques ont été proposées pour effectuer la vérification de la structure produite par les patrons. De cette façon, il est valide de considérer que ces patrons *permettent l'implémentation graduelle de mécanismes d'adaptation dans les interfaces graphiques* et que la structure de support est vérifiée et considérée comme valide.

7.1.2 Proposition de patrons de conception

Dans le cadre du projet de recherche, trois patrons de conception ont été proposés : *Moniteur*, *Proxy routeur* et *Composant adaptatif*. Ces patrons de conception sont issus d'un processus itératif où des préoccupations de l'adaptation ont été identifiées, des concepts essentiels relevés de la littérature puis implémentés dans le cadre de prototypes démonstratifs. De nouvelles préoccupations étaient par la suite relevées depuis les lectures effectuées pendant l'itération et des propositions d'amélioration émises. L'exécution de ces étapes, tirées directement des sous-objectifs, a permis le développement des patrons de conception et de leur implémentation de référence qui sont les sujets du premier article (voir chapitre 4). En plus, l'analyse des gains par rapport à une implémentation *ad hoc* a permis d'identifier que la séparation des préoccupations d'adaptation (stratégies d'adaptation, surveillance des données d'adaptation, reconfiguration) était accrue. De plus, les éléments reliés à l'adaptation étaient également eux-mêmes découplés des implémentations non adaptatives.

Pour les interfaces usagers graphiques qui constituent les cibles pour l'ajout de comportements adaptatifs dans le cadre du projet de recherche, cette dernière séparation permet ainsi de développement de *widgets* adaptatifs depuis ceux fournis par le *toolkit*. Ces *widgets*, les composants dans le contexte des interfaces usagers graphiques, deviennent alors des éléments pouvant être utilisés dans la conception d'une interface graphique avec l'éditeur graphique.

Suite à ces résultats, il est donc valide d'affirmer que les patrons de conception proposés sont issus de l'état de l'art en matière de structure pour l'adaptation puisque ces derniers sont le résultat d'un vaste recensement des concepts du domaine (voir section 4.2). Les patrons de conception proposés sont inspirés de ceux disponibles dans la littérature et en font la synthèse (voir les paragraphes « *Related patterns* » dans les définitions de patrons à la section 4.3). Finalement, les patrons remédient à certaines lacunes identifiées (ex. : la séparation de la modélisation à la conception et à l'exécution des données d'adaptation, l'utilisation collaborative de la substitution et de la paramétrisation). Leur utilisation représente un gain par rapport à l'implémentation *ad hoc* (voir section 4.5.2) dans le contexte des interfaces usagers graphiques.

7.1.3 Proposition de métriques

Dans le cadre du projet de recherche, deux métriques ont été proposées : la couverture modélisée de l'espace d'adaptation et la couverture effective de l'espace d'adaptation. Ces métriques visent un même but : quantifier la proportion des états atteignables par le système qu'un composant adaptatif supporte de façon explicite. Avant de pouvoir proposer ces métriques, des sous-objectifs d'analyse devaient être atteints au niveau de la recherche de métriques proposées dans la littérature. Aussi, un objectif visait l'identification d'éléments de mesure rendus disponibles par la structure induite par l'application des patrons de conception proposés dans le présent projet de recherche.

Grâce à une revue de littérature détaillée à la section 6.2.2, il a été constaté que les métriques existantes sont principalement axées sur la comparaison de systèmes adaptatifs ou sur la caractérisation de la performance de l'adaptation. Un manque notable se situe donc au niveau de l'évaluation de la structure supportant l'adaptation. Comme plusieurs applications contiennent des implémentations potentiellement différentes de mécanismes d'adaptation similaires, les différences structurelles pourraient biaiser la comparaison pour l'atteinte de buts d'adaptation et de performance. Ainsi, en évaluant séparément la structure, les résultats de l'évaluation des mécanismes d'adaptation pourraient être mieux interprétés lorsque les systèmes à l'étude partagent une structure considérée équivalente.

Au niveau des éléments de mesure tirés de la structure induite par les patrons, les espaces d'adaptation utilisés pour déclarer les états valides des composants adaptatifs ont été identifiés comme des éléments de mesure importants. En effet, ces derniers sont utilisés également pour définir l'espace contextuel du système où chaque dimension est constituée des valeurs possibles des paramètres à la conception et des moniteurs à l'exécution.

Un autre élément de mesure identifié est l'arbre de composition des composants adaptatifs. Comme les composants adaptatifs peuvent avoir des candidats de substitution, ces derniers peuvent être considérés comme les enfants du composant adaptatif dans un arbre. Comme chaque candidat de substitution peut à son tour être un composant adaptatif, ces relations peuvent être modélisées en un arbre de taille arbitraire. Les caractéristiques de cet arbre sont donc un autre élément de mesure.

Il existe des relations de dépendance qui ciblent un composant adaptatif et débutent depuis les moniteurs qui fournissent les valeurs des paramètres que ce dernier connait. Aussi, comme un moniteur peut baser ses valeurs sur un état interne au système, cet état peut être lié à des propriétés d'un composant, adaptatif ou non. Une telle situation crée alors un lien de dépendance depuis le moniteur vers le composant en question.

Ces derniers éléments de mesure ont été utilisés pour proposer des métriques de qualité potentielles (voir sections 5.1.2, 5.1.3 et 5.1.4). L'espace d'adaptation a permis la proposition des métriques citées précédemment et qui sont le sujet du chapitre 6. Leur implémentation est détaillée à la section 5.3. Finalement, elles sont appliquées à une application concrète et des actions correctives sont proposées à la section 6.5. Ainsi, il est valide de considérer que les objectifs de proposer et d'implémenter les métriques ainsi que de les utiliser pour analyser une application concrète sont atteints. L'atteinte des sous-objectifs est ainsi complète et il est également valide d'affirmer que l'objectif secondaire de proposition des métriques est atteint.

7.2 Choix techniques et compromis

Pour la réalisation du projet de recherche, certains choix ont dû être faits pour faciliter l'atteinte des objectifs et contrôler l'effort requis pour la réalisation des différents prototypes. La justification de ces choix est abordée pour chacun d'entre eux dans la présente section.

7.2.1 Langage et plate forme

Le langage choisi pour le projet de recherche est Python en version 3.4. Un tel langage a procuré de nombreux avantages dans le cadre du projet de recherche. D'abord, la focalisation du projet de recherche sur la portabilité des solutions implique que l'implémentation proposée des patrons de conception devait être également le plus portable possible. Comme Python est un langage interprété et qu'un interpréteur est disponible pour toutes les plates formes majeures (GNU/Linux, Windows et MacOS), ce langage remplissait ce critère. Un autre critère de base est que le langage doit supporter le POO, ce qui est le cas de Python. En plus, au niveau de l'implémentation des patrons eux-mêmes, le fait que Python soit un langage dynamique rend spécialement faciles la reconfiguration et la mise en place d'un *proxy*. Finalement, comme il ne s'agit pas d'un langage fortement typé (une variable peut accueillir n'importe quel type à n'importe quel moment), le développement de prototypes est facilité. Au niveau de l'assurance qualité, l'intégration des tests unitaires permet d'offrir un niveau acceptable de confiance aux utilisateurs envers l'implémentation des patrons.

7.2.2 *Librairie* pour l'adaptation

À l'origine, il avait été envisagé de baser les développements sur une librairie d'adaptation existante. La librairie envisagée était *ContextPy*, une implémentation de la méthode de

programmation orientée-contexte (*context-oriented programming*) proposée par Hirschfeld *et al.* [18]. La programmation orientée-contexte permet l'activation et la désactivation de couches dans lesquelles des méthodes sont réimplémentées. Ainsi, en fonction de l'état d'activation d'une couche, les méthodes appelées diffèrent. La mise en place de mécanismes d'adaptation est donc centralisée dans la logique d'activation et de désactivation des couches, tandis que des classes peuvent spécifier les comportements spécifiques à un contexte par le biais de la réimplémentation de leurs méthodes. Une analyse de l'utilisation de cette librairie a été faite et une version fonctionnant avec Python 3 a été produite.

La principale raison pour laquelle la programmation orientée-contexte a été mise de côté est que le principe des couches était trop contraignant. En effet, bien que la redirection des appels de fonction est un principe largement partagé dans le domaine de l'adaptation, une couche ne peut représenter qu'un état booléen d'une propriété ou d'une condition. Ainsi, pour couvrir un spectre plus large et complexe de contextes, beaucoup de couches seraient nécessaires et seulement des combinaisons booléennes sont possibles.

Ainsi, l'idée de développer des patrons de conception inspirés d'une large variété de travaux et où les méthodes d'adaptation seraient plus générales a poussé le développement d'une librairie indépendante. Le choix de ne pas se baser sur des développements existants a donc été fait pour contraindre le moins possible la mise en place d'artéfacts génériques et de se différencier des développements existants.

CHAPITRE 8

Conclusion

8.1 Sommaire

Le présent projet de recherche s'inscrit dans le domaine de l'adaptation dans les logiciels. Avec l'essor de l'Internet des Objects, les logiciels sont désormais présents dans une multitude d'appareils et l'automatisation de tâches de façon personnalisée. Cependant, mettre en place l'adaptation, particulièrement dans les interfaces personne-machine, reste un défi de taille. Il s'agit ainsi de chercher des techniques pour structurer un logiciel de sorte que l'adaptation puisse facilement et graduellement être ajoutée. De plus, il est nécessaire d'utiliser une façon adéquate de communiquer cette structure afin qu'elle soit utilisable dans une grande variété de logiciels, le plus indépendamment possible des technologies utilisées.

Les patrons de conception sont une formalisation d'une solution logicielle à un problème récurrent d'implémentation. Ces derniers sont largement utilisés dans le domaine du développement logiciel. Ainsi, l'objectif pour le présent projet de recherche était de trouver *quels sont les patrons de conception qui permettent l'implémentation graduelle de mécanismes d'adaptation dans les interfaces graphiques.* Pour atteindre cet objectif, une analyse en profondeur de la littérature sur le sujet de l'adaptation dans les logiciels a été faite afin de déceler les concepts et approches que partagent les travaux de l'état de l'art. Depuis ceux-ci, des patrons de conception ont été développés dans le contexte d'un processus itératif où des préoccupations de l'adaptation étaient abordées et implémentés depuis les solutions dans l'état de l'art dans des prototypes démonstratifs. Les approches utilisées pouvaient alors être critiquées et des lacunes identifiées de sorte que des améliorations puissent être apportées. Une fois que les préoccupations essentielles à la réalisation d'une application graphique furent couvertes, des patrons de conception ont été produits depuis une généralisation des solutions explorées.

Ces patrons de conception ont ensuite été implémentés dans une librairie de référence nommée *AdaptivePy.* Cette dernière fut utilisée afin de développer un prototype d'application graphique adaptative pour une étude de cas. Cette application est un jeu-questionnaire où les réponses passées influencent le nombre d'options proposées à l'utilisateur pour favoriser la polarisation des résultats. Une version de ce prototype utilisant une approche *ad hoc*

(sans la librairie) a été implémentée et utilisée pour comparaison. Les gains observés se situent principalement au niveau de la séparation des préoccupations, de la cohésion des méthodes, de la localisation des changements pour l'ajout de l'adaptation et de l'extensibilité. Ces propriétés induisent une meilleure maintenabilité. En plus, comme les composants adaptatifs formant l'application sont des instances autonomes, il est possible de les utiliser pour faire la conception de l'interface dans un éditeur graphique, ce qui n'était pas possible pour l'implémentation *ad hoc*. Pour les développeurs, cela représente un avantage, car le déroulement du travail (*workflow*) se rapproche plus de ce qui est habituel dans le domaine du développement d'interfaces graphiques.

Une fois les patrons formalisés et utilisés dans une application concrète, il a été réalisé que ces derniers ne suffisent pas à assurer une structure optimale ou valide à l'exécution de l'application. Pour mieux évaluer la structure induite par les patrons de conception, des métriques traditionnelles ont été analysées afin de déceler des analogies avec les éléments à évaluer d'un logiciel adaptatif. Par la suite, la structure induite elle-même a été analysée afin d'identifier des éléments de mesure pouvant servir à l'élaboration de métriques. Trois métriques potentielles ont été proposées et une d'entre elles, la couverture de l'espace d'adaptation, a été sélectionnée pour être formalisée.

En fonction des situations supportées par les composants adaptatifs, une situation n'étant supportée par aucun composant offrant un certain service induira un comportement indéfini au logiciel nécessitant ce service. Comme les composants adaptatifs doivent définir explicitement les situations qu'ils supportent sous forme « d'espace d'adaptation », une métrique visant l'évaluation de la couverture de cet espace a été proposée pour quantifier la proportion des situations atteignables supportées par un composant. De cette façon, il est possible pour un développeur de déterminer si un composant adaptatif peut induire des comportements indéfinis. Aussi, un développeur peut évaluer l'impact des actions qu'il a entreprises pour mieux couvrir les situations pouvant être rencontrées par le logiciel (augmentation, diminution ou aucun changement de la couverture). Deux versions de cette métrique ont été proposées : l'une basée sur les situations atteignables de façon théorique, soit en fonction de la modélisation faite du contexte, et l'autre sur les situations concrètement atteignables, soit en fonction des valeurs qui peuvent être rencontrées à l'exécution. Ces métriques ont été nommées respectivement « couverture modélisée de l'espace d'adaptation » et « couverture effective de l'espace d'adaptation ». Elles ont été formalisées dans un format similaire au standard ISO/IEC 25023 et appliquées dans le cadre d'une application typique. Les actions correctives sujettes à améliorer la couverture ont également été proposées et leur impact démontré.

Le projet de recherche a été organisé de sorte que les deux grands objectifs de recherche (la proposition de patrons de conception et la proposition de métriques pour la validation) soient abordés directement dans des articles séparés. Les détails additionnels fournis dans ce mémoire au niveau de la méthodologie et des approches connexes explorées permettent de démontrer la façon dont les travaux ont été réalisés. L'ensemble de ces éléments rencontrent donc la totalité des objectifs de recherche et les résultats obtenus forment une avancée notable de l'état de l'art au niveau des techniques de structuration de l'adaptation dans les logiciels de domaines variés.

Au final, la réponse à la question de recherche est donnée par la proposition des patrons de conception Moniteur, Proxy router et Composant adaptatif conjointement à la validation par les métriques de couverture modélisée et effective de l'espace d'adaptation. La structure de base mise en place par l'application des patrons de conception permet bien l'implémentation de différents mécanismes d'adaptation pendant le cycle de développement d'un logiciel adaptatif. Les mécanismes d'adaptation étant implémentés sous forme de stratégies, il est facile d'en comparer et d'en ajouter de façon graduelle.

8.2 Contributions

Les contributions originales au projet de recherche sont les patrons de conception et les métriques proposées. Une description plus détaillée de chacun est donnée dans cette section.

8.2.1 Patrons de conception

Les patrons de conception sont des formalisations de solutions à des problèmes récurrents, dans ce cas-ci au niveau de la structuration de préoccupation de l'adaptation. Les patrons proposés sont originaux, car ils sont une généralisation de solutions plus spécifiques à divers domaines. Ils sont composés de certains patrons existants plus primitifs, mais leur structure telle que présentée est originale. Surtout, leur utilisation commune pour mettre en place une structure alliant la paramétrisation et la substitution de composants dans une optique de collaboration est une avancée notable de l'état de l'art.

- **Moniteur :** Le patron de conception moniteur met en place une structure qui permet l'utilisation de données d'adaptation pour le contrôle de l'adaptation d'un ensemble de composants. Une approche basée sur des évènements permet d'émettre des notifications lorsque des changements de valeurs sont détectés. Ces notifications sont

envoyées aux composants intéressés et peuvent induire une adaptation. Les données d'adaptation sont fournies par des entités dédiées qui ont la responsabilité d'offrir une valeur dans un domaine connu, modélisé pour un type de données à la conception. Il est donc possible d'avoir plusieurs façons d'obtenir un type de données d'adaptation.

- **Proxy routeur :** Le patron de conception proxy routeur permet de rediriger les appels de fonction envers un composant concret en fonction d'une stratégie de routage. Ce patron est une utilisation collaborative des patrons proxy qui réalise un délégué pour un service et routeur qui redistribue un flux en fonction de stratégies spécifiques au domaine. Ce patron de conception met donc en place une structure supportant la substitution de composants et le découplage de mécanismes d'adaptation.

- **Composant adaptatif :** Le patron de conception composant adaptatif tire parti des deux autres patrons en ajoutant une structure générale pour un composant qui permet d'effectuer de l'adaptation par paramétrisation et par substitution. Le patron impose la divulgation par une interface commune d'un espace d'adaptation qui stipule les situations contextuelles supportées par le composant par l'une ou l'autre des méthodes d'adaptation. Aussi, un lien de dépendance sur un gestionnaire d'évènements pour l'obtention de notification de changement de données d'adaptation est imposé. De cette façon, l'ensemble des composants partageant ce gestionnaire peuvent participer de façon collaborative à un plan d'adaptation puisque ces derniers s'entendent alors sur l'état du système à tout moment. Le fait qu'un candidat de substitution tel que défini dans le patron Proxy routeur puisse être un composant adaptatif, il est également possible de créer des hiérarchies d'une taille arbitraire de composants adaptatifs. Cette structure se prête particulièrement bien aux applications graphiques.

8.2.2 Métriques d'évaluation

Bien que trois métriques générales aient été proposées, une formalisation de seulement une d'entre elles en deux versions spécifiques a été effectuée dans le cadre du présent projet de recherche. Cependant, il est adéquat de les mentionner ici puisqu'elles sont, en effet, originales dans le contexte des logiciels adaptatifs.

Profondeur de l'arbre de substitution

Comme un arbre peut être généré depuis les liens de candidature de substitution pour les composants adaptatifs, la profondeur de cet arbre peut être considéré comme une mesure de la complexité d'exécution de l'adaptation. Plus le nombre de niveaux de l'arbre est

grand, plus l'exécution d'une reconfiguration, par exemple, a un potentiel de complexité élevé.

Stabilité de l'adaptation

Les relations de dépendance entre les moniteurs et les composants adaptatifs peuvent donner lieu à un cycle d'adaptation potentiellement infini. Une telle situation est indésirable, or le nombre de tels cycles dans une architecture peut être considérée une mesure de la stabilité d'adaptation. La valeur idéale est 0, soit l'absence complète de cycles.

Couverture de l'espace d'adaptation (*retenue*)

L'espace d'adaptation est l'ensemble des situations contextuelles que supporte un composant. Ainsi, la proportion de cet ensemble sur l'ensemble des situations atteignable par le système, sa « couverture », est un indicateur de la présence de situations qui ne sont pas supportées par ce composant. Il s'agit également d'un indicateur de la spécialisation d'un composant ; un composant supportant toutes les situations est « général », tandis qu'un composant supportant peu de situations est « spécifique ». On peut donc comparer deux composants et indiquer que l'un est plus spécifique ou général que l'autre.

8.2.3 Métriques spécifiques formalisées

La métrique de couverture de l'espace d'adaptation a été spécialisée en deux versions. Ces versions sont synthétisées ici.

Couverture *modélisée* de l'espace d'adaptation

Cette couverture est basée sur l'espace contextuel que forment les valeurs possibles des **paramètres**. Les paramètres définissent un type de donnée d'adaptation ainsi que son domaine de valeurs *théorique*. Ainsi, un composant adaptatif peut être développé pour supporter un ensemble de situations qui pourraient être rencontrées dans n'importe lequel des environnements d'exécution. La métrique indique la proportion sur l'ensemble de ces situations potentielles, neutre de l'environnement d'exécution, que supporte un composant. Cette propriété permet d'être calculée à la conception sans connaissance des environnements de déploiement.

Couverture *effective* de l'espace d'adaptation

Cette couverture est basée sur l'espace contextuel que forment les valeurs possibles des **moniteurs**. Un moniteur a la responsabilité de fournir une valeur parmi celles définies par un et un seul paramètre. Comme ces valeurs sont fournies à l'exécution, la méthode

de surveillance et de collecte des données peut être dépendante de l'environnement de déploiement. Les moniteurs offrent donc un domaine de valeurs *pratique* pour l'ensemble des données d'adaptations. La métrique indique la proportion sur l'ensemble des situations qui pourraient être rencontrées à l'exécution que supporte un composant. Comme ni l'identité des moniteurs ni même les valeurs qu'ils peuvent fournir ne sont pas fixes à la conception, cette métrique ne peut être calculée qu'à l'exécution. Il est cependant possible de calculer la métrique à la conception en spécifiant un ensemble de moniteurs.

8.3 Travaux futurs

Les travaux du présent projet de recherche sont concentrés sur l'identification de concepts fondamentaux pour la structure d'un logiciel adaptatif et des méthodes pour mettre en place cette structure dans une grande variété de logiciels, mais particulièrement pour les applications graphiques. Cependant, certains éléments n'ont pas été entièrement couverts ou simplement omis pour préciser les éléments de recherche ou faciliter l'obtention de résultats. Le sujet de cette section est donc de clarifier les éléments qui n'ont pas été couverts par le présent projet de recherche et d'indiquer des pistes pour les travaux futurs.

8.3.1 Éléments à approfondir dans le cadre de recherches futures

Le présent projet de recherche atteint tous les objectifs fixés initialement dans la définition de projet. Cependant, l'attention particulière du projet de recherche a été sur le développement d'une structure pour le développement d'interfaces graphiques. Ainsi, en focalisant sur ce domaine, certains concepts inapplicables comme le support de transactions simultanées envers plusieurs délégués (pour les systèmes distribués en réseau) ont été mis de côté. L'étendue du domaine de l'adaptation dans les logiciels étant beaucoup plus large que les limites fixées pour le projet de recherche, certains aspects pourraient être approfondis dans le futur.

Par exemple, comme la structure de base mise en place par l'application de patrons de conception est neutre de mécanismes d'adaptation, il serait intéressant de démontrer par le biais de plusieurs prototypes l'utilisation de différents mécanismes avec une même structure. Par exemple, l'utilisation d'un moteur d'inférence par règles pour l'adaptation est un mécanisme populaire et son implémentation dans le cadre de la structure de base pourrait servir de guide pour des développeurs désirant l'utiliser.

Dans le premier article, des métriques étaient utilisées de façon qualitative pour effectuer une analyse des gains en qualité apportés par l'utilisation des patrons de conception. Dans des travaux futur, il pourrait être intéressant de quantifier ces métriques (cohésion des méthodes, séparation des préoccupations) et d'automatiser leur calcul afin qu'elles puissent être utilisées plus facilement dans le cadre de prototypes complexes.

Dans le cas des métriques, il serait intéressant de formaliser les autres métriques potentielles qui ont été proposées (*profondeur de l'arbre de substitution, stabilité de l'adaptation*) dans le format ISO/IEC 25023. Un élément qui aurait pu être plus couvert par rapport à la métrique formalisée est son utilisation dans une vaste gamme d'applications, soit des applications simples et d'autres très complexes. Ainsi, dans un projet futur, il serait possible de mieux évaluer la difficulté d'application des actions correctives et de proposer des pistes pour accompagner les développeurs dans leur application. Il a déjà été indiqué qu'une quantité importante de paramètres (plus de trois) empêchait la représentation graphique de l'espace contextuel et donc la détermination et l'application des actions correctives proposées. Aussi, les impacts possiblement négatifs (effets secondaires) qu'aurait l'application de chacune des actions correctives pourraient être explorés. Il serait notamment utile de mieux comprendre les impacts de choisir une action corrective par rapport à une autre en fonction du manque de couverture détecté.

8.3.2 Nouvelles perspectives de recherche

Étant donné que le projet de recherche visait spécifiquement les interfaces graphiques, il serait intéressant d'explorer les avenues apportées par l'adaptation dans ce domaine. Plusieurs travaux existants visent à mettre en place des mécanismes d'adaptation dans les interfaces graphiques (« *plastic ui* » [9, 11, 28]), mais aucune structure de base n'est mise de l'avant pour supporter une grande variété de mécanismes. Un aspect qui pourrait être étudié est la façon dont une interface peut adapter son comportement en fonction du médium d'entrée (souris, clavier, stylet, écran capacitif). En théorie, l'utilisation des patrons de conception permettrait de mettre en place une structure adéquate pour le développement d'une telle interface, mais les éléments de conception dans le domaine des interfaces personne-machine ne sont pas spécifiquement adressés dans la présente recherche. Ainsi, une avenue serait de créer une banque de contrôles pour les interfaces graphiques qui s'adaptent au médium d'entrée. La fonctionnalité de prévisualisation de l'adaptation qui est rendue possible dans l'éditeur graphique permettrait de faciliter la conception de tels composants. Un type de composant complexe à développer pour ce domaine de contexte

serait des « *layouts* », soit des *widgets* qui disposent d'autres *widgets* dans une région de l'interface.

Un autre aspect qui serait à explorer est l'expansion de la librairie AdaptivePy pour supporter par défaut les transitions d'états intermédiaires pour composants lors du transfert d'état. Ainsi, les transactions concurrentielles pourraient être supportées et la librairie serait mieux adaptée aux systèmes distribués en réseau. Une extension de la librairie spécifique à l'utilisation pour la création d'interfaces pourrait également être créée pour simplifier son utilisation pour ce genre d'applications.

Finalement, une perspective de recherche intéressante serait d'automatiser l'application des patrons de conception afin qu'une application puisse être générée depuis des requis haut niveau d'adaptation et une modélisation de l'espace contextuel. Ainsi, il serait en théorie possible de générer une base d'application qui pourrait alors être utilisée comme un *framework*. Cela pourrait permettre de régler en partie le problème de spécialisation des *frameworks* et permettrait de minimiser les problèmes liés à une mauvaise implémentation des patrons. Dans le cas des interfaces graphiques, il serait intéressant de considérer si la majeure partie de l'interface pourrait être générée automatiquement depuis la spécification d'un appareil, plate forme, type d'utilisateur et méthodes d'entrée et d'affichage.

LISTE DES RÉFÉRENCES

[1] Aubin, J., Bayen, A. et Saint-Pierre, P. (2011). *Viability Theory : New Directions*. Modern Birkhèauser classics, Springer Berlin Heidelberg.

[2] Berkane, M. L., Seinturier, L. et Boufaida, M. (2015). Using variability modelling and design patterns for self-adaptive system engineering : Application to smart-home. *Int. J. Web Eng. Technol.*, volume 10, numéro 1, p. 65–93.

[3] Bezold, M. et Minker, W. (2011). *Adaptive multimodal interactive systems*. Springer Science & Business Media.

[4] Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V. et Stefani, J.-B. (2006). The fractal component model and its support in java. *Software : Practice and Experience*, volume 36, numéro 11-12, p. 1257–1284.

[5] Calinescu, R., Ghezzi, C., Kwiatkowska, M. et Mirandola, R. (2012). Self-adaptive software needs quantitative verification at runtime. *Commun. ACM*, volume 55, numéro 9, p. 69–77.

[6] Chang, F. et Karamcheti, V. (2001). A framework for automatic adaptation of tunable distributed applications. *Cluster Computing*, volume 4, numéro 1, p. 49–62.

[7] Chen, W.-K., Hiltunen, M. A. et Schlichting, R. D. (2001). Constructing adaptive software in distributed systems. Dans *Distributed Computing Systems, 2001. 21st International Conference on.* p. 635–643.

[8] Cheng, B. H. C., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D. et Whittle, J. (2009). *Software Engineering for Self-Adaptive Systems*, chapitre Software Engineering for Self-Adaptive Systems : A Research Roadmap. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 1–26.

[9] Collignon, B., Vanderdonckt, J. et Calvary, G. (2008). Model-driven engineering of multi-target plastic user interfaces. Dans *Fourth International Conference on Autonomic and Autonomous Systems (ICAS'08)*. p. 7–14.

[10] Corsaro, A., Schmidt, D. C., Klefstad, R. et O'Ryan, C. (2002). Virtual component - a design pattern for memory-constrained embedded applications. Dans *in Proceedings*

of the Ninth Conference on Pattern Language of Programs (PLoP.

[11] Coutaz, J. (2010). User interface plasticity : Model driven engineering to the limit ! Dans *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*. EICS '10. ACM, New York, NY, USA, p. 1–8.

[12] Gamma, E., Helm, R., Johnson, R. et Vlissides, J. (1995). *Design Patterns : Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[13] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B. et Steenkiste, P. (2004). Rainbow : Architecture-based self-adaptation with reusable infrastructure. *Computer*, volume 37, numéro 10, p. 46–54.

[14] Georgiadis, I., Magee, J. et Kramer, J. (2002). Self-organising software architectures for distributed systems. Dans *Proceedings of the First Workshop on Self-healing Systems*. WOSS '02. ACM, New York, NY, USA, p. 33–38.

[15] Gomaa, H., Hashimoto, K., Kim, M., Malek, S. et Menascé, D. A. (2010). Software adaptation patterns for service-oriented architectures. Dans *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. ACM, New York, NY, USA, p. 462–469.

[16] Gomaa, H. et Hussein, M. (2004). Software reconfiguration patterns for dynamic evolution of software architectures. Dans *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*. p. 79–88.

[17] Hinchey, M. G. et Sterritt, R. (2006). Self-managing software. *Computer*, volume 39, numéro 2, p. 107–109.

[18] Hirschfeld, R., Costanza, P. et Nierstrasz, O. (2008). Context-oriented programming. Dans *Journal of Object Technology, March-April 2008, ETH Zurich*, Citeseer. volume 7.

[19] Holvoet, T., Weyns, D. et Valckenaers, P. (2009). Patterns of delegate mas. Dans *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*. p. 1–9.

[20] IBM (2005). *An architectural blueprint for autonomic computing* (Rapport technique). IBM Corporation.

[21] ISO/IEC (2016). *Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – Measurement of system and software*

product quality (Standard 25023).

[22] Kaddoum, E., Raibulet, C., Georgé, J.-P., Picard, G. et Gleizes, M.-P. (2010). Criteria for the evaluation of self-* systems. Dans *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '10. ACM, New York, NY, USA, p. 29–38.

[23] Kang, P., Heffner, M., Mukherjee, J., Ramakrishnan, N., Varadarajan, S., Ribbens, C. et Tafti, D. K. (2007). The adaptive code kitchen : Flexible tools for dynamic application composition. Dans *2007 IEEE International Parallel and Distributed Processing Symposium*. p. 1–8.

[24] Lemos, R., Giese, H., Müller, H. A., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Goeschka, K. M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, R., Schmerl, B., Smith, D. B., Sousa, J. P., Tamura, G., Tahvildari, L., Villegas, N. M., Vogel, T., Weyns, D., Wong, K. et Wuttke, J. (2013). Software engineering for self-adaptive systems : A second research roadmap. Dans de Lemos, R., Giese, H., Müller, H. A. et Shaw, M., *Software Engineering for Self-Adaptive Systems II, volume 7475*. Springer-Verlag, p. 1–32.

[25] Liu, H. et Parashar, M. (2006). Accord : a programming framework for autonomic applications. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, volume 36, numéro 3, p. 341–352.

[26] Liu, Y., Xu, C. et Cheung, S. (2013). Afchecker : Effective model checking for context-aware adaptive applications. *Journal of Systems and Software*, volume 86, numéro 3, p. 854 – 867.

[27] Longchamps, S. et Gonzalez-Rubio, R. (2017). Design patterns for addition of adaptive behavior in graphical user interfaces. Dans *Proceedings of the Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications*.

[28] Majrashi, K., Hamilton, M. et Uitdenbogerd, A. (2015). Multiple user interfaces and cross-platform user experience : Theoretical foundations. Dans *CCSEA 2015*, AIRCC Publishing Corporation. p. 43–57.

[29] Malek, S., Beckman, N., Mikic-Rakic, M. et Medvidovic, N. (2005). *A Framework for Ensuring and Improving Dependability in Highly Distributed Systems*. Springer

Berlin Heidelberg, Berlin, Heidelberg, p. 173–193.

[30] Mannava, V. et Ramesh, T. (2012). Multimodal pattern-oriented software architecture for self-optimization and self-configuration in autonomic computing system using multi objective evolutionary algorithms. Dans *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*. ICACCI '12. ACM, New York, NY, USA, p. 1236–1243.

[31] Maurel, Y., Diaconescu, A. et Lalanda, P. (2010). Ceylon : A service-oriented framework for building autonomic managers. Dans *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*. p. 3–11.

[32] McCann, J. A. et Huebscher, M. C. (2004). *Evaluation Issues in Autonomic Computing*. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 597–608.

[33] Menasce, D. A., Sousa, J. a. P., Malek, S. et Gomaa, H. (2010). Qos architectural patterns for self-architecting software systems. Dans *Proceedings of the 7th International Conference on Autonomic Computing*. ICAC '10. ACM, New York, NY, USA, p. 195–204.

[34] Montague, K., Hanson, V. L. et Cobley, A. (2012). Designing for individuals : Usable touch-screen interaction through shared user models. Dans *Proceedings of the 14th International ACM SIGACCESS Conference on Computers and Accessibility*. ASSETS '12. ACM, New York, NY, USA, p. 151–158.

[35] Motti, V. G. et Vanderdonckt, J. (2013). A computational framework for context-aware adaptation of user interfaces. Dans *IEEE 7th International Conference on Research Challenges in Information Science (RCIS)*. p. 1–12.

[36] Peissner, M., Schuller, A. et Spath, D. (2011). A design patterns approach to adaptive user interfaces for users with special needs. Dans *Proceedings of the 14th International Conference on Human-computer Interaction : Design and Development Approaches - Volume Part I*. HCII'11. Springer-Verlag, Berlin, Heidelberg, p. 268–277.

[37] Perez-Palacin, D., Mirandola, R. et Merseguer, J. (2014). On the relationships between qos and software adaptability at the architectural level. *Journal of Systems and Software*, volume 87, p. 1 – 17.

[38] Raibulet, C., Arcelli Fontana, F., Capilla, R. et Carrillo, C. (2016). An overview on quality evaluation of self-adaptive systems. Dans Mistrik, I., Ali, N., Kazman, R.,

Grundy, J. et Schmerl, B., *Managing Trade-Offs in Adaptable Software Architectures*, 1re édition. Morgan Kaufmann, Boston, p. 325 – 352.

[39] Ramirez, A. J. (2008). *Design patterns for developing dynamically adaptive systems*. Mémoire de maîtrise, Michigan State University.

[40] Ramirez, A. J. et Cheng, B. H. (2010). Design patterns for developing dynamically adaptive systems. Dans *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ACM. p. 49–58.

[41] Rossi, G., Gordillo, S. et Lyardet, F. (2005). Design patterns for context-aware adaptation. Dans *2005 Symposium on Applications and the Internet Workshops (SAINT 2005 Workshops)*. p. 170–173.

[42] Rushby, J. (2008). *Runtime Certification*. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 21–35.

[43] Salehie, M. et Tahvildari, L. (2009). Self-adaptive software : Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, volume 4, numéro 2, p. 14.

[44] Tamura, G., Villegas, N. M., Müller, H. A., Sousa, J. P., Becker, B., Karsai, G., Mankovskii, S., Pezzè, M., Schäfer, W., Tahvildari, L. et Wong, K. (2013). *Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, p. 108–132.

[45] Taylor, R. N., Medvidovic, N., Anderson, K. M., Whitehead Jr, E. J., Robbins, J. E., Nies, K. A., Oreizy, P. et Dubrow, D. L. (1996). A component-and message-based architectural style for gui software. *Software Engineering, IEEE Transactions on*, volume 22, numéro 6, p. 390–406.

[46] Villegas, N. M., Müller, H. A., Tamura, G., Duchien, L. et Casallas, R. (2011). A framework for evaluating quality-driven self-adaptive software systems. Dans *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. SEAMS '11. ACM, New York, NY, USA, p. 80–89.

[47] Weyns, D., Schmerl, B., Grassi, V., Malek, S., Mirandola, R., Prehofer, C., Wuttke, J., Andersson, J., Giese, H. et Göschka, K. M. (2013). On patterns for decentralized control in self-adaptive systems. Dans *Software Engineering for Self-Adaptive Systems II*. Springer, p. 76–107.

[48] Zhang, J. et Cheng, B. H. C. (2006). Model-based development of dynamically adaptive software. Dans *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. ACM, New York, NY, USA, p. 371–380.