Une enquête sur le test de mutation: principes, avantages, limites et orientations futures

Moez Krichen

Une enquête sur le test de mutation: principes, avantages, limites et orientations futures

Moez Krichen

Laboratoire ReDCAD, Université de Sfax, Tunisie
moez.krichen@redcad.org

Résumé. En raison de sa fiabilité et de son efficacité, l'approche du test de mutation (Mutation Testing - MT) pour le test de logiciels a gagné en popularité ces dernières années. Cette revue de la littérature pertinente fournit un résumé de MT, y compris son histoire, son état actuel, ses applications, ses avantages, ses inconvénients et ses perspectives d'avenir. Au départ, l'article définit MT et explique sa pertinence dans le test de logiciels. L'article se poursuit ensuite en définissant MT et en expliquant en quoi il diffère des autres méthodes de test de logiciels. Certaines des techniques discutées ci-dessus comprennent le statement MT, le decision MT, le condition MT, le data MT et le object-oriented MT. Ensuite, l'étude se penche sur les avantages et les inconvénients de l'intégration de MT dans le processus de test de logiciels, ainsi que sur l'état actuel des méthodologies et de la technologie de MT. Nous analysons également les avantages et les inconvénients de MT par rapport aux autres méthodes de test de logiciels. Dans ses dernières sections, l'article met en évidence les difficultés du MT existant et propose des solutions potentielles pour la recherche future, telles que l'intégration de MT avec d'autres stratégies de test de logiciels.

1 Introduction

Le logiciel étant présent dans presque tous les aspects de notre vie quotidienne, sa qualité et sa fiabilité sont devenues une préoccupation majeure pour les organisations et les utilisateurs. Les exemples d'applications logicielles dans les secteurs tels que les transports (13; 14), la santé (25; 24), la blockchain et d'autres, soulignent l'importance de la qualité et de la fiabilité du logiciel dans la vie quotidienne. Les erreurs ou les défauts dans le logiciel peuvent avoir des conséquences graves, allant de pertes financières aux risques pour la sécurité et la vie privée. C'est pourquoi le test logiciel est devenu une étape cruciale dans le développement des applications logicielles.

Les tests de logiciels, qui visent à assurer la qualité et la fiabilité des systèmes logiciels, constituent un élément crucial du développement de logiciels (6; 12; 30). Au fil des ans, de nombreuses méthodes de tests de logiciels ont été créées, chacune présentant des avantages et des inconvénients uniques (29). Le test de mutation (Mutation Testing

- MT) est l'une de ces méthodes qui a récemment gagné en popularité en tant que moyen fiable d'évaluer la qualité des cas de test (43; 4).

MT est une technique de test basée sur les défauts qui consiste à créer un ensemble de mutants, qui sont des versions du programme contenant un ou plusieurs défauts (49; 47; 44). L'objectif de MT est de mesurer l'efficacité des cas de test en calculant le score de mutation, qui est le pourcentage de mutants tués par la suite de tests (35; 18; 28). Si un mutant n'est tué par aucun cas de test, il est considéré comme équivalent à un défaut réel qui n'a pas été détecté, et le score de mutation est réduit (42).

Les mutants sont créés en apportant de petites modifications au code du programme d'origine. L'objectif est d'introduire des défauts dans le programme qui sont représentatifs des erreurs de programmation courantes, telles que :

- Changer un opérateur conditionnel (par exemple, remplacer "==" par "!=").
- Changer un opérateur arithmétique (par exemple, remplacer "+" par "-").
- Supprimer une instruction ou un bloc de code.
- Échanger deux instructions ou blocs de code.
- Ajouter une instruction ou un bloc de code.
- Les opérateurs relationnels sont remplacés par leur complément (par exemple, < par >=).

Cette étude de revue fournit un aperçu de la MT, en couvrant ses principes, ses types, ses innovations, ses applications industrielles, ses avantages, ses limites, ses défis ouverts et ses orientations futures. Le document commence par définir la MT et souligner son importance dans les tests de logiciels (15; 46). Il explique ensuite les principes de base de la MT et la compare à d'autres techniques de tests de logiciels (11; 9; 3; 33; 48).

Le document couvre également différentes classes et types de MT, notamment la MT de déclaration, de décision, de condition, de données et orientée objet. Il discute des avancées récentes dans les techniques et les outils de MT, ainsi que des défis et des avantages de la mise en œuvre de la MT dans l'industrie. Les avantages et les limites de la MT sont analysés, ainsi qu'une comparaison avec d'autres techniques de tests de logiciels.

Vers la fin du document, les défis liés à la MT sont identifiés, et des suggestions pour des recherches futures sont proposées. Celles-ci comprennent l'intégration de la MT avec d'autres méthodologies de tests de logiciels pour surmonter ses limites. Dans l'ensemble, cette étude de revue fournit un aperçu complet de la MT, qui sera précieux à la fois pour les praticiens et les chercheurs en tests de logiciels. Elle met en évidence le potentiel de la MT en tant que technique de test utile et identifie des domaines pour des améliorations futures.

2 Principe de MT

MT est une technique de test logiciel puissante qui consiste à créer un ensemble de mutants, ou variations du programme original, qui contiennent une ou plusieurs anomalies. Le principe de base de MT est d'évaluer l'efficacité des cas de test en mesurant leur capacité à identifier ces anomalies. Le processus de création de mutants

consiste à appliquer un ou plusieurs opérateurs de mutation au code du programme, ce qui entraîne la création de nouvelles versions du programme contenant des anomalies. Les opérateurs de mutation peuvent être basés sur une variété de critères, tels que le changement de la valeur d'une variable, l'échange d'opérateurs ou la modification de déclarations conditionnelles.

2.1 Idée de base

L'objectif de MT est d'évaluer la qualité des cas de test en mesurant le score de mutation, qui est le pourcentage de mutants détectés par la suite de test. Un score de mutation élevé indique que la suite de test est efficace pour détecter les anomalies du programme, tandis qu'un score de mutation faible indique que la suite de test peut être insuffisante et nécessite une amélioration. MT peut être utilisé pour compléter d'autres techniques de test, telles que le test fonctionnel et le test de couverture de code, afin d'améliorer la qualité et la fiabilité des systèmes logiciels.

MT est basé sur l'hypothèse que les anomalies dans un programme sont similaires aux mutations dans le matériel génétique, dans le sens où elles peuvent être considérées comme des modifications aléatoires qui affectent le comportement du programme. Les mutations peuvent être utilisées pour simuler différents types d'anomalies, telles que des erreurs de syntaxe, des erreurs logiques et des erreurs de données. En introduisant ces anomalies dans le programme, MT fournit une évaluation plus réaliste de l'efficacité des cas de test que d'autres techniques de test.

2.2 Définition formelle

Soit P un programme, et soit T un ensemble de cas de test conçus pour tester P. Soit M un ensemble de mutants qui sont créés en appliquant un ou plusieurs opérateurs de mutation au code de P. Chaque mutant m dans M est une variation de P qui contient une ou plusieurs anomalies.

L'objectif de MT est d'évaluer l'efficacité de T en mesurant sa capacité à détecter les anomalies dans les mutants de M. Pour chaque mutant m dans M, soit $f(m)$ une fonction qui représente le comportement de m. La fonction $f(m)$ prend en entrée les mêmes données que P et produit les sorties correspondantes de m.

Le score de mutation, $SM(T)$, est défini comme :

$$SM(T) = \frac{(|M| - |K(T, M)|)}{|M|} * 100\%$$

où $|M|$ est le nombre total de mutants dans M, et $|K(T, M)|$ est le nombre de mutants dans M qui sont tués par T. Un mutant m est considéré comme étant tué par T s'il existe au moins un cas de test dans T qui produit une sortie différente pour m que pour P.

Intuitivement, le score de mutation mesure la capacité de T à détecter les anomalies dans le programme. Si T est efficace, il devrait être capable de détecter la plupart des anomalies dans les mutants, ce qui se traduit par un score de mutation élevé. Si T est inefficace, il ne réussira pas à détecter de nombreuses anomalies, ce qui se traduira par un score de mutation faible.

2.3 Étapes pour la génération et l'exécution de mutants

Il existe plusieurs étapes impliquées dans la génération de mutants pour MT, notamment l'identification des opérateurs de mutation, l'application de ces opérateurs au programme original, la compilation et l'exécution des mutants, ainsi que l'évaluation de l'efficacité de la suite de test pour détecter les anomalies. Ces étapes fournissent un cadre général pour la génération de mutants en MT et peuvent être adaptées en fonction de l'environnement de test et du programme à tester. Plus de détails sur ces étapes sont fournis ci-dessous :

1. **Identifier les opérateurs de mutation :** Identifier les opérateurs de mutation qui seront utilisés pour générer les mutants. Différents opérateurs de mutation peuvent être utilisés pour générer différents types de mutants, tels que des mutants faibles ou forts.
2. **Appliquer les opérateurs de mutation :** Appliquer les opérateurs de mutation au programme original pour générer un ensemble de mutants. Pour le MT traditionnel, un seul opérateur de mutation est appliqué au programme original à la fois pour générer des mutants. Pour d'autres types de MT, tels que le MT faible ou fort, plusieurs opérateurs de mutation peuvent être appliqués au programme original pour générer un plus grand ensemble de mutants.
3. **Compiler et exécuter les mutants :** Compiler et exécuter les mutants pour déterminer leur comportement. Les mutants doivent être compilés et exécutés dans le même environnement que le programme original, en utilisant les mêmes données d'entrée et les mêmes conditions d'exécution. Le comportement de chaque mutant doit être comparé au comportement du programme original pour déterminer si le mutant introduit une anomalie.
4. **Évaluer l'efficacité de la suite de test :** Évaluer l'efficacité de la suite de test pour détecter les anomalies introduites par les mutants. La suite de test doit être exécutée sur chaque mutant, et le nombre de mutants détectés par la suite de test doit être enregistré. L'efficacité de la suite de test peut être mesurée à l'aide de métriques telles que le score de mutation, qui est le pourcentage de mutants détectés par la suite de test.

Ces étapes fournissent un cadre général pour la génération de mutants en MT. En fonction de l'environnement de test et du programme à tester, des étapes supplémentaires ou des modifications de ces étapes peuvent être nécessaires.

2.4 Différentes classes de MT

Il existe différentes classes de MT, telles que les MT forts et faibles ainsi que les MT conventionnels. Dans les MT conventionnels, un seul opérateur de mutation est utilisé à la fois pour créer des mutants. Le processus de création de mutants pour les MT faibles nécessite la combinaison de différents opérateurs de mutation. La création de mutations qui sont fonctionnellement identiques au programme original mais avec une implémentation différente fait partie des MT forts. Il existe d'autres types de MT, moins populaires, tels que les MT comparables, les tests basés sur les schémas de mutants et

les tests basés sur le comportement. Nous donnons ci-dessous plus d'informations sur ces différentes classifications :

- **MT traditionnel :** Le MT traditionnel est la classe de MT la plus largement utilisée, où un seul opérateur de mutation est appliqué au programme original à la fois pour générer un ensemble de mutants. Un opérateur de mutation est une règle ou un algorithme qui modifie le programme original en introduisant une erreur (par exemple, en changeant un opérateur relationnel ou en supprimant une instruction conditionnelle). L'objectif du MT traditionnel est d'évaluer l'efficacité d'un ensemble de tests donné pour détecter les erreurs introduites par les opérateurs de mutation. Le MT traditionnel est informatiquement coûteux, car il nécessite la génération et l'exécution d'un grand nombre de mutants.
- **MT faible :** Le MT faible est une variation du MT traditionnel, où plusieurs opérateurs de mutation sont appliqués au programme original pour générer des mutants. L'idée derrière le MT faible est qu'il peut augmenter le nombre de mutants générés et réduire le nombre de tests requis pour évaluer l'ensemble de tests, tout en restant informatiquement faisable. Le MT faible peut également aider à détecter des erreurs qui sont manquées par le MT traditionnel.
- **MT fort :** Le MT fort est une autre variation du MT traditionnel, où les mutants générés sont fonctionnellement équivalents au programme original, mais avec une implémentation différente. L'objectif du MT fort est d'évaluer la capacité d'un ensemble de tests à détecter des erreurs dans l'implémentation du programme, plutôt que simplement dans sa syntaxe. Le MT fort est plus difficile à implémenter que le MT traditionnel ou faible, car il nécessite la génération de mutants qui sont fonctionnellement équivalents au programme original.
- **MT équivalent :** Le MT équivalent est une classe moins courante de MT qui implique la génération de mutants qui sont sémantiquement équivalents au programme original, mais avec une syntaxe différente. L'objectif du MT équivalent est d'évaluer la capacité d'un ensemble de tests à détecter des erreurs liées à la structure et à la syntaxe du programme.
- **Tests basés sur les schémas de mutants :** Les tests basés sur les schémas de mutants sont une autre classe moins courante de MT qui implique la génération de mutants basés sur un schéma qui définit la structure et la syntaxe des mutants. L'objectif des tests basés sur les schémas de mutants est de générer des mutants qui sont plus pertinents pour la structure et la syntaxe du programme, et de réduire le nombre de mutants non pertinents qui doivent être générés.
- **MT basé sur le comportement :** Le MT basé sur le comportement est une classe plus récente de MT qui implique la génération de mutants en modifiant le comportement du programme, plutôt que sa syntaxe. L'objectif du MT basé sur le comportement est de générer des mutants qui sont plus représentatifs du comportement réel du programme, et de réduire le nombre de mutants non pertinents qui doivent être générés.

En fonction des objectifs du processus de test et des caractéristiques du logiciel testé, chaque classe de MT présente des avantages et des inconvénients distincts et peut être utilisée.

3 Comparaison avec d'autres types de tests

En plus du MT, il existe plusieurs autres types de tests que les développeurs peuvent utiliser pour assurer la qualité et la fiabilité de leur logiciel. Chaque type de test a ses propres forces et faiblesses, et les développeurs doivent choisir le type de test approprié en fonction des exigences spécifiques du logiciel. Dans cette comparaison, nous nous concentrerons sur le MT et le comparerons à d'autres types de tests pour mettre en évidence ses avantages et limites uniques.

- **Tests unitaires (17) :** Ils consistent à tester les composants individuels du code pour s'assurer qu'ils fonctionnent correctement. L'accent est mis sur l'identification des bugs dans de petits morceaux de code isolés. Bien que les tests unitaires fassent partie intégrante de toute stratégie de test, ils peuvent ne pas être suffisants pour détecter toutes les erreurs dans le code. Contrairement au MT, qui évalue l'efficacité des cas de test eux-mêmes en introduisant des erreurs dans le code, les tests unitaires ne vérifient que le bon fonctionnement des composants individuels.
- **Tests d'intégration (10) :** Ils consistent à tester les interactions entre différents composants pour s'assurer qu'ils fonctionnent ensemble correctement. L'accent est mis sur l'identification des bugs qui se produisent lorsque les composants sont combinés. Bien que les tests d'intégration soient importants, ils peuvent ne pas détecter toutes les erreurs qui découlent des composants individuels. Le MT peut identifier les erreurs à la fois dans les composants individuels et dans leurs interactions avec d'autres composants.
- **Tests système (50) :** Ils consistent à tester l'ensemble du système pour s'assurer qu'il répond aux exigences et fonctionne comme prévu. L'accent est mis sur l'identification des bugs qui découlent du système dans son ensemble. Bien que les tests système soient importants, ils peuvent ne pas détecter toutes les erreurs qui découlent des composants individuels ou de leurs interactions. Le MT peut compléter les tests système en identifiant les faiblesses des cas de test eux-mêmes.
- **Tests d'acceptation (7) :** Ils consistent à tester le logiciel pour s'assurer qu'il répond aux exigences et satisfait les besoins des utilisateurs. L'accent est mis sur l'évaluation du logiciel du point de vue des utilisateurs finaux. Bien que les tests d'acceptation soient importants, ils peuvent ne pas détecter toutes les erreurs qui découlent des composants individuels ou de leurs interactions. Le MT peut compléter les tests d'acceptation en identifiant les faiblesses des cas de test eux-mêmes.
- **Tests d'injection de fautes (34) :** Ils consistent à introduire intentionnellement des erreurs dans le code ou l'environnement pour observer le comportement du système dans ces conditions. L'accent est mis sur l'évaluation de la résilience du système aux erreurs imprévues. Bien que les tests d'injection de fautes puissent être utiles pour identifier les faiblesses du système, ils peuvent ne pas être aussi efficaces que le MT pour évaluer l'efficacité des cas de test. Les tests d'injection de fautes sont plus axés sur l'identification des erreurs déjà connues dans le système, tandis que le MT est conçu pour identifier les faiblesses des cas de test eux-mêmes.

— **Tests basés sur des modèles (22; 21; 19; 26; 20; 27) :** Ils consistent à créer des modèles du système et à utiliser ces modèles pour générer des cas de test. Les tests basés sur des modèles peuvent être utiles pour générer un grand nombre de tests et vérifier que le système répond aux exigences. Bien que les tests basés sur des modèles puissent être efficaces, ils peuvent ne pas détecter toutes les erreurs qui se produisent dans le code réel. Le MT peut compléter les tests basés sur des modèles en identifiant les faiblesses des cas de test eux-mêmes. Cependant, le développement de modèles précis peut être chronophage et peut nécessiter des compétences et des outils spécialisés.
— **Tests de sécurité (23) :** Ils visent à identifier les vulnérabilités et à garantir que le logiciel est protégé contre les attaques malveillantes. Les tests de sécurité sont essentiels pour garantir que le logiciel est sûr et protégé contre les menaces externes. Bien que le MT puisse aider à identifier les erreurs de sécurité dans les cas de test, il ne remplace pas les tests de sécurité spécifiques qui sont nécessaires pour garantir la sécurité globale du logiciel.
— **Tests de charge (39; 38; 40; 41) :** Ils visent à évaluer la performance du logiciel sous différentes charges de travail et à identifier les limites du système. Les tests de charge sont essentiels pour garantir que le logiciel peut fonctionner de manière fiable dans des conditions de charge élevée ou variable. Bien que le MT puisse aider à identifier les erreurs dans les cas de test liées à la charge, il ne remplace pas les tests de charge spécifiques qui sont nécessaires pour garantir la performance globale du logiciel.
— **Tests de régression (31)** Ils visent à garantir que les modifications apportées au logiciel n'ont pas introduit de nouveaux bugs ou altéré le comportement du logiciel existant. Les tests de régression sont essentiels pour garantir que le logiciel reste fiable et conforme aux exigences initiales. Bien que le MT puisse aider à identifier les erreurs dans les cas de test liées à la régression, il ne remplace pas les tests de régression spécifiques qui sont nécessaires pour garantir la fiabilité globale du logiciel.

En conclusion, le MT est un type de test unique qui complète les autres types de tests en identifiant les faiblesses des cas de test eux-mêmes. Bien que le MT ne remplace pas les autres types de tests, il peut améliorer leur efficacité en identifiant les faiblesses des cas de test et en améliorant leur qualité globale. Le MT peut être utilisé dans le cadre d'une stratégie de test complète pour garantir la qualité et la fiabilité du logiciel.

4 Différents types de MT

Les MT peuvent être classés en différents types en fonction des critères utilisés pour générer les mutants :

— **MT de déclaration :** Les MT de déclaration impliquent la modification de déclarations individuelles dans le programme en introduisant des défauts. Cela peut être fait en modifiant les valeurs des constantes, des variables ou des opérateurs, ou en supprimant ou en insérant des déclarations. L'objectif des MT de déclaration est d'évaluer l'efficacité des cas de test pour détecter des défauts dans les déclarations individuelles.

— **MT de décision :** Les MT de décision impliquent la modification du flux de contrôle du programme en modifiant les conditions de branchement des instructions if et des boucles. Cela peut être fait en modifiant les opérateurs relationnels, logiques ou les expressions booléennes utilisées dans les conditions. L'objectif des MT de décision est d'évaluer l'efficacité des cas de test pour détecter des défauts dans le flux de contrôle du programme.
— **MT de condition :** Les MT de condition sont une version plus fine des MT de décision qui impliquent la modification des expressions booléennes utilisées dans les conditions des instructions if et des boucles. Les mutations peuvent être créées en modifiant les opérateurs booléens, les opérandes des opérateurs ou la négation des opérateurs. L'objectif des MT de condition est d'évaluer l'efficacité des cas de test pour détecter des défauts dans les expressions booléennes utilisées dans les conditions.
— **MT de données :** Les MT de données impliquent la modification des données d'entrée utilisées pour exécuter le programme. Cela peut être fait en modifiant les valeurs des constantes, des variables ou des structures de données. L'objectif des MT de données est d'évaluer l'efficacité des cas de test pour détecter des défauts dans les données d'entrée du programme.
— **MT orienté objet :** Les MT orienté objet impliquent la modification de la structure du programme, telle que la modification de la hiérarchie d'héritage ou des appels de méthodes entre objets. Les mutations peuvent être créées en modifiant les appels de méthodes, en modifiant les modificateurs d'accès des méthodes et des champs, ou en modifiant les relations d'héritage. L'objectif des MT orienté objet est d'évaluer l'efficacité des cas de test pour détecter des défauts dans la conception et la mise en œuvre orientées objet du programme.

Dans l'ensemble, chaque type de MT offre une perspective différente sur l'efficacité des cas de test et peut être utile pour identifier différents types de défauts dans le programme. Le choix du type dépend des objectifs des tests et des caractéristiques du programme. En utilisant une combinaison de différents types de MT, une évaluation complète de la qualité des cas de test et de la fiabilité des systèmes logiciels peut être obtenue.

5 Applications industrielles de MT

MT a été appliqué avec succès dans diverses applications industrielles et réelles pour améliorer la qualité des logiciels. Voici quelques exemples :

— **Kernel Linux :** La principale contribution de l'étude présentée dans (2) a été de démontrer l'utilité de l'analyse de mutation dans les grands systèmes, notamment sur le module RCU du noyau Linux. Les auteurs ont adapté des techniques existantes pour répondre aux exigences de calcul et aux nombreux faux positifs généralement associés à l'analyse de mutation. L'étude a montré que l'analyse de mutation était capable d'identifier des lacunes dans le harnais de test RCU et a mis au jour deux bogues masqués dans le module RCU. Les auteurs soutiennent que MT devrait être utilisé plus largement dans la pratique

car il peut être un outil précieux pour mesurer l'exhaustivité et la qualité d'un jeu de tests et identifier les lacunes dans même les modules bien testés.

— **Applications Web :** MUTTA, un outil automatisé pour MT des applications Web en test de bout en bout (E2E), a été proposé par les auteurs de (32). Les fichiers sources du serveur de l'application Web cible sont modifiés par MUTTA, qui exécute ensuite la suite de tests E2E contre les applications Web modifiées et recueille les résultats des tests. Le programme génère un grand nombre de mutants, utilise des données de couverture pour prendre en compte uniquement les mutants exécutés, et enregistre les résultats de la suite de tests de manière acceptable. Le processus MT des suites de tests E2E peut être automatisé à l'aide de MUTTA, et il peut être utilisé pour évaluer les suites de tests E2E, selon une étude de cas que les auteurs ont réalisée pour comparer deux méthodologies de test E2E Web. L'article présente une nouvelle technique pour évaluer automatiquement l'efficacité des suites de tests pour améliorer la qualité des applications Web.
— **Programmes Spark :** L'étude présentée dans (8) a proposé TRANSMUT-Spark, un outil pour automatiser la MT du code de traitement de données volumineuses des programmes Spark. L'étude met en évidence la complexité de la programmation parallèle de données volumineuses et la nécessité de combiner intelligemment les fonctions intégrées de Spark dans les programmes pour exploiter les ressources de calcul nécessaires au traitement de données volumineuses et éviter des pertes de sortie importantes. MT est utilisée pour tester les applications Spark, et TRANSMUT-Spark automatise la génération de mutants, l'exécution des tests et l'analyse d'adéquation. La publication couvre également l'étendue et les limites de l'outil sur la base d'études de validation. L'article présente une nouvelle méthode pour évaluer les ensembles de tests et trouver des erreurs de code dans les programmes Spark.
— **Applications distribuées :** MuTomVo, un cadre MT pour les applications distribuées dans des environnements simulés, a été présenté dans (5). L'étude souligne les contraintes de test des applications distribuées dans des systèmes fortement distribués pendant le développement et propose d'utiliser des plates-formes de simulation pour modéliser une large gamme de configurations de systèmes distribués et exécuter ces applications dans le système modélisé. Une suite de tests peut être exécutée contre l'ensemble de modèles modifiés pour évaluer sa capacité de détection d'erreurs à l'aide du cadre proposé. L'article explique la mise en œuvre du cadre dans MuTomVo et donne une étude de cas de trois applications fonctionnant dans divers systèmes distribués pour démontrer sa praticité. Dans l'ensemble, la recherche propose un cadre unique qui peut améliorer la qualité des applications distribuées en offrant un mécanisme fiable et automatisé pour évaluer la détection d'erreurs des suites de tests dans des situations simulées.
— **Projets logiciels à grande échelle :** L'étude (45) a proposé HadoopMutator, une plateforme MT basée sur le cloud qui utilise MapReduce pour accélérer la création et le test de mutants. L'article propose une solution pour améliorer la MT automatisée et fournir une solution évolutive pour les projets logiciels à

grande échelle. Dans deux situations d'application, le système suggéré améliore la MT automatisée d'un facteur de 10. En traitant le code source comme des données, les frameworks centrés sur les données plus récents offrent des options de réutilisation supplémentaires, selon le rapport. L'article présente un cadre MT qui aide à augmenter la qualité des projets logiciels en évaluant les méthodologies de test plus rapidement et de manière plus évolutive.

— **Programmes Java :** Il existe plusieurs systèmes de MT différents disponibles pour Java (1; 16), notamment :
  — **MuJava (μJava) :** MuJava est un outil conçu spécifiquement pour la MT d'applications Java (37; 36). L'outil crée des mutants en utilisant une variété d'opérateurs de mutation sur le code source d'origine, notamment en modifiant les appels de méthode, en changeant les opérateurs arithmétiques ou logiques et en supprimant des déclarations. MuJava prend en charge une variété de techniques de MT, telles que la MT forte, faible et conventionnelle. En plus de supporter la génération automatisée de cas de test et l'analyse de couverture, MuJava offre également un ensemble complet d'outils pour la génération, l'exécution et l'analyse de mutants.
  — **Jester :** Jester est un outil de MT populaire pour Java, C et C++. Pour créer des mutants, il modifie les opérateurs relationnels ou supprime les instructions conditionnelles. Jester exécute ensuite le jeu de tests sur chaque mutant et calcule un score de mutation, pourcentage de mutations découvertes. Jester est extrêmement adaptable, permettant aux utilisateurs de définir des opérateurs de mutation et des niveaux de mutation de code. Il est rapide et évolutif, ce qui permet aux utilisateurs de tester des bases de code énormes de manière efficace. Jester a été utilisé dans plusieurs projets de développement de logiciels pour trouver des bugs ignorés par les tests standard.
  — **Simple Jester :** Simple Jester est un outil de MT développé pour être une alternative plus légère à l'outil Jester original. Comme Jester, Simple Jester est conçu pour fonctionner avec du code Java et applique des opérateurs de mutation pour générer un ensemble de mutants. Il exécute ensuite le jeu de tests sur chaque mutant et calcule un score de mutation. Simple Jester est conçu pour être facile à utiliser et pour fournir une rétroaction rapide sur l'efficacité d'un jeu de tests. Il dispose d'une interface de ligne de commande simple et peut être intégré à des outils d'intégration continue comme Jenkins. Bien qu'il ne soit peut-être pas aussi riche en fonctionnalités que d'autres outils de MT, Simple Jester peut être un bon choix pour les projets qui nécessitent une solution de MT légère et efficace.
  — **Jumble :** Jumble est un outil de MT conçu pour être hautement personnalisable et extensible. Il peut être utilisé avec des bases de code Java, .NET et Ruby et offre une grande variété d'opérateurs de mutation. Jumble permet également aux utilisateurs de créer leurs propres opérateurs de mutation et de configurer l'outil pour appliquer des mutations de manière sélective en fonction de critères spécifiques, tels que la complexité du code ou la couverture de test. Jumble dispose d'une interface en ligne de commande et

peut être intégré à des outils de construction tels qu'Ant, Maven ou Gradle. Jumble a été utilisé dans une variété de projets de développement de logiciels et s'est avéré efficace pour identifier des défauts qui sont ignorés par d'autres méthodes de test.

— **Javalanche :** Javalanche est un outil de MT conçu pour être hautement parallélisable et évolutif. Il applique des opérateurs de mutation au code Java et génère un ensemble de mutants, puis exécute le jeu de tests sur chaque mutant et calcule un score de mutation. Javalanche est conçu pour bien fonctionner avec de grandes bases de code et peut être exécuté en parallèle sur plusieurs machines. Il fournit également des rapports détaillés sur les résultats du processus de test, y compris des informations sur les mutants qui ont été tués par le jeu de tests et ceux qui ne l'ont pas été. Javalanche peut être intégré à des outils de construction tels qu'Ant et Maven et fournit également une interface web pour visualiser les résultats de la MT. Javalanche a été utilisé dans une variété de projets de développement de logiciels et s'est avéré efficace pour identifier des défauts qui sont ignorés par d'autres méthodes de test.

— **PIT :** PIT (abréviation de "Pitest") est un outil de MT Java populaire. Il génère une collection de mutations, exécute le jeu de tests sur chaque mutant et calcule un score de mutation. PIT permet aux utilisateurs de choisir des opérateurs de mutation, des classes à tester et d'autres fonctionnalités. Les utilisateurs peuvent également utiliser les rapports et visualisations de PIT pour comprendre les résultats des tests. La "MT incrémentale", qui aide à accélérer la MT sur de grandes bases de code, est l'une des fonctionnalités de PIT. PIT est généralement connecté à des technologies de construction telles que Maven ou Gradle et peut être exécuté automatiquement dans un flux de travail d'intégration continue. PIT a été utilisé dans de nombreux projets de développement de logiciels pour trouver des défauts ignorés par les méthodes de test conventionnelles.

Ces exemples démontrent l'efficacité de la MT pour identifier les faiblesses de la suite de tests et améliorer la qualité, la fiabilité et les performances globales du logiciel dans diverses applications industrielles.

6 Avantages et Limitations

Comme toute technique de test, le test de mutation (Mutation Testing - MT) a ses avantages et ses limites.

6.1 Avantages

L'un des principaux avantages de la MT est sa capacité à identifier les faiblesses et les lacunes de la suite de test. En introduisant de petits changements dans le code, la MT peut évaluer l'efficacité de la suite de test dans la détection de défauts et identifier les zones où la suite de test peut être améliorée. Cela fait de la MT un outil précieux pour améliorer la qualité globale du logiciel. De plus, la MT peut détecter des défauts

qui ne peuvent pas être détectés par d'autres techniques de test. En introduisant de petits changements dans le code, la MT peut révéler des défauts que d'autres techniques ne peuvent pas détecter, tels que les conditions aux limites ou les défauts de logique complexes. Cela fait de la MT un complément précieux aux autres techniques de test, améliorant l'efficacité globale du processus de test. Enfin, la MT peut évaluer la qualité de la suite de test elle-même. En générant et en exécutant un grand nombre de mutants, la MT peut déterminer l'efficacité et la fiabilité de la suite de test dans la détection de défauts. Cela fait de la MT un outil précieux pour garantir la qualité et la fiabilité du processus de test.

6.2 Limitations

Cependant, il y a aussi des limites à l'utilisation de la MT. L'une des principales limites est les exigences computationnelles et le temps nécessaire pour exécuter le grand nombre de mutants générés.

Bien que la MT soit un moyen efficace d'évaluer l'adéquation d'une suite de test, elle comporte des coûts significatifs en termes d'argent et de ressources. L'un des coûts principaux de la MT est les ressources computationnelles nécessaires pour exécuter les tests. Cette technique implique la génération d'un grand nombre de mutants, chacun représentant une version légèrement modifiée du logiciel original, et l'exécution de la suite de test contre chacun de ces mutants. À mesure que le nombre de mutants augmente, la quantité de puissance de calcul et le temps d'exécution requis pour le processus de test augmentent de manière exponentielle. Cela peut entraîner des coûts supplémentaires en termes de matériel et de consommation d'énergie.

En plus des exigences computationnelles, la MT peut également être coûteuse en termes de ressources humaines. Le processus de génération et d'analyse de mutants nécessite souvent des professionnels qualifiés du test de logiciel, qui sont très demandés et qui commandent des salaires élevés. De plus, lorsque la suite de test est jugée insuffisante pour détecter des mutants, les développeurs peuvent être amenés à consacrer du temps et des efforts supplémentaires à l'amélioration de la suite de test, ce qui peut encore augmenter le coût du projet.

De plus, le coût des outils et des licences de MT ne doit pas être négligé. Bien qu'il existe des frameworks de MT open source disponibles, de nombreuses organisations choisissent d'investir dans des outils commerciaux offrant des fonctionnalités plus avancées et un meilleur support. Ces outils peuvent être coûteux, et le coût peut être prohibitif pour les petites organisations ou les projets ayant des budgets serrés.

De plus, la nature chronophage de la MT peut entraîner des retards dans le cycle de développement de logiciel. Le processus de génération, d'exécution et d'analyse de mutants peut prendre une quantité considérable de temps, surtout s'il y a un grand nombre de mutants et des suites de test complexes. Cela peut entraîner des coûts supplémentaires car le temps de mise sur le marché d'un produit logiciel est prolongé, ce qui peut entraîner une perte de revenus et de parts de marché.

En outre, les mutations qui n'introduisent pas réellement de défauts dans le code peuvent être difficiles à distinguer de celles qui le font, ce qui entraîne un grand nombre de faux positifs. Cela peut rendre difficile l'interprétation des résultats du test et peut entraîner un gaspillage d'efforts dans l'investigation des faux positifs. Enfin, la MT peut

ne pas être efficace dans la détection de certains types de défauts, tels que ceux liés aux conditions aux limites ou à la performance. Dans de tels cas, d'autres techniques de test peuvent être plus appropriées.

En résumé, la MT est une technique puissante pour évaluer la qualité d'une suite de test de logiciel, mais elle comporte des coûts importants en termes de ressources computationnelles, humaines, d'outillage et de temps. En tant que tel, les organisations doivent peser soigneusement les avantages de la MT par rapport à ces coûts pour déterminer si la technique est un investissement rentable pour leur projet spécifique.

7 Directions Futures

Pour répondre à certaines des limitations précédemment mentionnées, les chercheurs ont proposé diverses techniques telles que la mutation sélective, où seul un sous-ensemble de mutants est généré et exécuté, et la mutation d'ordre supérieur, qui introduit des mutations plus complexes qui peuvent être plus efficaces pour détecter les défauts. De plus, les progrès récents en matière de puissance de calcul et de techniques de parallélisation ont rendu la MT plus faisable pour une utilisation dans des systèmes plus importants. Ci-dessous, nous fournissons une brève explication de ces techniques et de la façon dont elles ont contribué à rendre la MT plus faisable pour les systèmes plus importants.

- **Mutation sélective :** La mutation sélective vise à réduire le nombre de mutants générés et exécutés en se concentrant sur un sous-ensemble de mutations considérées comme plus efficaces pour détecter les défauts. En sélectionnant seulement un ensemble plus petit de mutants représentatifs, le coût computationnel de la MT peut être considérablement réduit. Certaines stratégies pour la sélection de mutants comprennent :
  - **Échantillonnage aléatoire :** Sélection d'un sous-ensemble aléatoire de mutants.
  - **Techniques basées sur le regroupement :** Regroupement des mutants en fonction de leur similarité et sélection d'un représentant de chaque groupe.
  - **Sélection basée sur l'opérateur :** Priorisation de certains opérateurs de mutation qui ont été démontrés comme étant plus efficaces pour détecter les défauts.
- **Mutation d'ordre supérieur :** La mutation d'ordre supérieur implique la création de mutants avec des modifications multiples et simultanées. Ces mutations complexes peuvent conduire à une détection de défauts plus réaliste et efficace, car elles peuvent mieux représenter les types de défauts qui se produisent dans le développement de logiciels en temps réel. Bien que les mutants d'ordre supérieur puissent être plus coûteux en termes de calcul à générer et à analyser, ils ont le potentiel d'améliorer considérablement les taux de détection des défauts.
- **Progrès en matière de puissance de calcul et de parallélisation :** À mesure que la puissance de calcul s'est améliorée, il est devenu plus faisable d'exécuter la MT sur des systèmes plus importants. De plus, les techniques de parallélisation peuvent également aider à répartir la charge de travail de la MT

sur plusieurs processeurs ou machines, accélérant ainsi le processus. Certaines stratégies de parallélisation comprennent :

- **Parallélisme au niveau de la tâche :** Division de l'ensemble du processus MT en tâches plus petites qui peuvent être exécutées simultanément.
- **Parallélisme au niveau des données :** Partitionnement des données d'entrée (par exemple, la suite de tests ou la base de code) à traiter de manière indépendante par différents processeurs ou machines.
- **Parallélisme hybride :** Combinaison du parallélisme au niveau de la tâche et du parallélisme au niveau des données pour optimiser l'utilisation des ressources et accélérer la MT.

En conclusion, la mutation sélective, la mutation d'ordre supérieur et les progrès en matière de puissance de calcul et de techniques de parallélisation ont rendu la MT plus faisable pour une utilisation dans des systèmes plus importants. Par conséquent, les développeurs et les testeurs peuvent mieux évaluer la qualité de leur logiciel et identifier les défauts plus efficacement.

8 Conclusion

Pour conclure, cet article a fourni un aperçu complet de la MT, une technique qui s'est avérée efficace pour améliorer la qualité des applications logicielles. Nous avons discuté de l'idée de base et de la définition formelle de la MT, comparé cette technique à d'autres types de tests, et exploré différents types de MT et leurs applications industrielles. Nous avons également examiné le coût de la MT et identifié ses avantages, limitations et futures directions. En particulier, nous avons souligné le potentiel de recherche et de développement supplémentaire dans le domaine de la MT, y compris l'automatisation et l'intégration de cette technique avec d'autres méthodes de test.

En résumé, la MT est un ajout précieux à toute stratégie de test, offrant des avantages en termes d'identification de défauts difficiles à détecter et d'augmentation de la confiance dans la fiabilité du logiciel. Ses limites, telles que le coût computationnel élevé et la nécessité de testeurs expérimentés, peuvent être surmontées grâce à une planification et une mise en œuvre rigoureuses. En fin de compte, l'utilisation de la MT peut conduire à un logiciel de meilleure qualité, une plus grande satisfaction des clients et une plus grande réussite commerciale.

En regardant vers l'avenir, il y a beaucoup de potentiel pour une recherche et un développement supplémentaires dans le domaine de la MT. Une zone prometteuse d'investigation est l'automatisation de la MT, ce qui réduirait le temps et l'effort nécessaires pour générer et analyser les mutants. Une autre direction pour les travaux futurs est l'intégration de la MT avec d'autres techniques de test, comme le fuzz testing, pour améliorer l'efficacité de la stratégie de test globale. De plus, il est nécessaire de mener davantage de recherches sur la scalabilité de la MT, en particulier pour les applications logicielles complexes et volumineuses. Enfin, le développement de meilleurs outils et techniques pour l'analyse des résultats de la MT peut aider à maximiser les avantages de cette technique. En abordant ces défis et autres, le domaine de la MT peut continuer à croître et à évoluer, offrant une valeur encore plus grande aux développeurs et aux utilisateurs de logiciels.

En fin de compte, la MT est un ajout précieux à toute stratégie de test et devrait être considérée par les développeurs et les testeurs de logiciels cherchant à améliorer la fiabilité et la qualité de leurs applications. Avec une planification et une mise en œuvre rigoureuses, la MT peut aider à identifier les défauts et à garantir que les applications logicielles répondent aux normes les plus élevées de qualité et de fiabilité.

Références

[1] Sara Tarek ElSayed Abbas, Rohayanti Hassan, Shahliza Abd Halim, Shahreen Kasim, and Rohaizan Ramlan. Investigation on java mutation testing tools. *JOIV : International Journal on Informatics Visualization*, 6(2-2) :455–462, 2022.

[2] Iftekhar Ahmed, Carlos Jensen, Alex Groce, and Paul E. McKenney. Applying mutation analysis on kernel test suites : An experience report. In *2017 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 110–115, 2017.

[3] Bernhard K Aichernig, Florian Lorber, and Dejan Ničković. Time for mutants—model-based mutation testing with timed automata. In *Tests and Proofs : 7th International Conference, TAP 2013, Budapest, Hungary, June 16-20, 2013. Proceedings 7*, pages 20–38. Springer, 2013.

[4] Karam Al Kontar, Joumana Naji, Freddy Demiane, Salma Sobeh, and Ramzi Haraty. A survey on mutation testing approaches. In *2019 IEEE CHILEAN Conference on Electrical, Electronics Engineering, Information and Communication Technologies (CHILECON)*, pages 1–7. IEEE, 2019.

[5] Pablo C Cañizares, Alberto Núñez, and Mercedes G Merayo. Mutomvo : Mutation testing framework for simulated cloud and hpc environments. *Journal of Systems and Software*, 143 :187–207, 2018.

[6] Jarbele Coutinho, Wilkerson Andrade, and Patricia Machado. A survey of requirements engineering and software testing practices in agile teams. In *Proceedings of the 7th Brazilian Symposium on Systematic and Automated Software Testing*, pages 9–18, 2022.

[7] Lívia Fernanda de Oliveira, Cássio Leonardo Rodrigues, and Renato de Freitas Bulcão-Neto. Characterizing the software acceptance testing and the inclusion of people with disabilities by means of a systematic mapping. *IEEE Latin America Transactions*, 21(1) :35–46, 2023.

[8] João Batista de Souza Neto, Anamaria Martins Moreira, Genoveva Vargas-Solar, and Martin A Musicante. Transmut-spark : Transformation mutation for apache spark. *Software Testing, Verification and Reliability*, 32(8) :e1809, 2022.

[9] Sandra Camargo Pinto Ferraz Fabbri, José Carlos Maldonado, Paulo Cesar Masiero, Márcio Eduardo Delamaro, and E Wong. Mutation testing applied to validate specifications based on petri nets. In *Formal Description Techniques VIII : Proceedings of the IFIP TC6 Eighth International Conference on Formal Description Techniques, Montreal, Canada, October 1995*, pages 329–337. Springer, 1996.

[10] Mark Grechanik and Gurudev Devanla. Mutation integration testing. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 353–364. IEEE, 2016.

[11] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, (4) :371–379, 1982.

[12] Timo Hynninen, Jussi Kasurinen, Antti Knutas, and Ossi Taipale. Software testing : Survey of the industry practices. In *2018 41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1449–1454. IEEE, 2018.

[13] Rateb Jabbar, Noora Fetais, Mohamed Kharbeche, Moez Krichen, Kamel Barkaoui, and Mohammed Shinoy. Blockchain for the internet of vehicles : how to use blockchain to secure vehicle-to-everything (v2x) communication and payment ? *IEEE Sensors Journal*, 21(14) :15807–15823, 2021.

[14] Rateb Jabbar, Mohammed Shinoy, Mohamed Kharbeche, Khalifa Al-Khalifa, Moez Krichen, and Kamel Barkaoui. Urban traffic monitoring and modeling system : An iot solution for enhancing road safety. In *2019 international conference on internet of things, embedded systems and communications (iintec)*, pages 13–18. IEEE, 2019.

[15] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5) :649–678, 2010.

[16] Manju Khari. Comparison and validation of mutation testing tools based on java language. In *Optimization of Automated Software Testing Using Meta-Heuristic Techniques*, pages 13–29. Springer, 2022.

[17] Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Simon and Schuster, 2020.

[18] Marinos Kintis, Mike Papadakis, Andreas Papadopoulos, Evangelos Valvis, Nicos Malevris, and Yves Le Traon. How effective are mutation testing tools ? an empirical analysis of java mutation testing tools with manual analysis and real faults. *Empirical Software Engineering*, 23 :2426–2463, 2018.

[19] Moez Krichen. *Model-based testing for real-time systems*. PhD thesis, PhD thesis, PhD thesis, Universit Joseph Fourier (December 2007), 2007.

[20] Moez Krichen. A formal framework for conformance testing of distributed real-time systems. In *International Conference On Principles Of Distributed Systems*, pages 139–142. Springer, 2010.

[21] Moez Krichen. A formal framework for black-box conformance testing of distributed real-time systems. *International Journal of Critical Computer-Based Systems*, 3(1-2) :26–43, 2012.

[22] Moez Krichen. *Contributions to model-based testing of dynamic and distributed real-time systems*. PhD thesis, École Nationale d'Ingénieurs de Sfax (Tunisie), 2018.

[23] Moez Krichen, Omar Cheikhrouhou, Mariam Lahami, Roobaea Alroobaea, and Afef Jmal Maâlej. Towards a model-based testing framework for the security of

internet of things for smart city applications. In *Smart Societies, Infrastructure, Technologies and Applications : First International Conference, SCITA 2017, Jeddah, Saudi Arabia, November 27–29, 2017, Proceedings 1*, pages 360–365. Springer International Publishing, 2018.

[24] Moez Krichen, Afef Jmal Maâlej, and Mariam Lahami. A model-based approach to combine conformance and load tests : an ehealth case study. *International Journal of Critical Computer-Based Systems*, 8(3-4) :282–310, 2018.

[25] Moez Krichen, Seifeddine Mechti, Roobaea Alroobaea, Elyes Said, Parminder Singh, Osamah Ibrahim Khalaf, and Mehedi Masud. A formal testing model for operating room control system using internet of things. *Computers, Materials & Continua*, 66(3) :2997–3011, 2021.

[26] Moez Krichen and Stavros Tripakis. State identification problems for timed automata. In *TestCom*, volume 5, pages 175–191, 2005.

[27] Moez Krichen and Stavros Tripakis. Interesting properties of the real-time conformance relation tioco. In *Theoretical Aspects of Computing-ICTAC 2006 : Third International Colloquium, Tunis, Tunisia, November 20-24, 2006. Proceedings 3*, pages 317–331. Springer Berlin Heidelberg, 2006.

[28] Janice S Kwon, Angelica M Gutierrez-Barrera, Diana Young, Charlotte C Sun, Molly S Daniels, Karen H Lu, and Banu Arun. Expanding the criteria for brca mutation testing in breast cancer survivors. *Journal of clinical oncology*, 28(27) :4214–4220, 2010.

[29] Mariam Lahami and Moez Krichen. A survey on runtime testing of dynamically adaptable and distributed systems. *Software Quality Journal*, 29(2) :555–593, 2021.

[30] Mariam Lahami, Moez Krichen, Mariam Bouchakwa, and Mohamed Jmaiel. Using knapsack problem model to design a resource aware test architecture for adaptable and distributed systems. In *Testing Software and Systems : 24th IFIP WG 6.1 International Conference, ICTSS 2012, Aalborg, Denmark, November 19-21, 2012. Proceedings 24*, pages 103–118. Springer Berlin Heidelberg, 2012.

[31] Mariam Lahami, Moez Krichen, and Mohamed Jmaïel. Runtime testing approach of structural adaptations for dynamic and distributed systems. *International Journal of Computer Applications in Technology*, 51(4) :259–272, 2015.

[32] Maurizio Leotta, Davide Paparella, and Filippo Ricca. Mutta : a novel tool for e2e web mutation testing. *Software Quality Journal*, pages 1–22, 2023.

[33] Paul Lochhead, Aya Kuchiba, Yu Imamura, Xiaoyun Liao, Mai Yamauchi, Reiko Nishihara, Zhi Rong Qian, Teppei Morikawa, Jeanne Shen, Jeffrey A Meyerhardt, et al. Microsatellite instability and braf mutation testing in colorectal cancer prognostication. *Journal of the National Cancer Institute*, 105(15) :1151–1156, 2013.

[34] Jakub Lojda, Richard Panek, Jakub Podivinsky, Ondrej Cekan, Martin Krcma, and Zdenek Kotasek. Testing embedded software through fault injection : Case study on smart lock. In *2021 IEEE 22nd Latin American Test Symposium (LATS)*, pages 1–6. IEEE, 2021.

[35] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. Deepmutation : Mutation testing of deep learning systems. In *2018 IEEE 29th international symposium on software reliability engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[36] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava : an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2) :97–133, 2005.

[37] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : a mutation system for java. In *Proceedings of the 28th international conference on Software engineering*, pages 827–830, 2006.

[38] Afef Jmal Maâlej, Manel Hamza, Moez Krichen, and Mohamed Jmaiel. Automated significant load testing for ws-bpel compositions. In *2013 IEEE sixth international conference on software testing, verification and validation workshops*, pages 144–153. IEEE, 2013.

[39] Afef Jmal Maâlej, Moez Krichen, and Mohamed Jmaiel. Conformance testing of ws-bpel compositions under various load conditions. In *2012 IEEE 36th annual computer software and applications conference*, pages 371–371. IEEE, 2012.

[40] Afef Jmal Maâlej, Moez Krichen, and Mohamed Jmaiel. Model-based conformance testing of ws-bpel compositions. In *2012 IEEE 36th annual computer software and applications conference workshops*, pages 452–457. IEEE, 2012.

[41] Afef Jmal Maâlej, Mariam Lahami, Moez Krichen, and Mohamed Jmaïel. Distributed and resource-aware load testing of ws-bpel compositions. In *ICEIS (2)*, pages 29–38, 2018.

[42] A Jefferson Offutt, Roy P Pargas, Scott V Fichter, and Prashant K Khambekar. Mutation testing of software using mimd computer. In *ICPP (2)*, pages 257–266, 1992.

[43] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances : an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier, 2019.

[44] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing : Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 47–53. IEEE, 2018.

[45] Iman Saleh and Khaled Nagi. Hadoopmutator : A cloud-based mutation testing framework. In *Software Reuse for Dynamic Systems in the Cloud and Beyond : 14th International Conference on Software Reuse, ICSR 2015, Miami, FL, USA, January 4-6, 2015. Proceedings 14*, pages 172–187. Springer, 2014.

[46] David Schuler and Andreas Zeller. Javalanche : Efficient mutation testing for java. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 297–298, 2009.

[47] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sér-

gio Lopes de Souza. A systematic review on search based mutation testing. *Information and Software Technology*, 81 :19–35, 2017.

[48] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. Automated test generation and mutation testing for alloy. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 264–275. IEEE, 2017.

[49] W Eric Wong and Aditya P Mathur. Reducing the cost of mutation testing : An empirical study. *Journal of Systems and Software*, 31(3) :185–196, 1995.

[50] Muhammad Nouman Zafar. *Model-based System Testing of Safety-Critical Embedded Software*. Malardalen University (Sweden), 2022.