**Titre : Une approche pour le test non-fonctionnel et la configuration automatiques des générateurs**

**Résumé en français :**

Les techniques émergentes de l’ingénierie dirigée par les modèles et de la programmation générative ont permis la création de plusieurs générateurs (générateurs de code et compilateurs). Ceux-ci sont souvent utilisés afin de faciliter le développement logiciel et automatiser le processus de génération de code à partir des spécifications abstraites. En effet, les utilisateurs sont devenus capables de synthétiser rapidement des artefacts logiciels pour une large gamme de plateformes logicielles et matérielles. De plus, les générateurs modernes comme les compilateurs C, sont devenus hautement configurables, offrant de nombreuses options de configuration à l'utilisateur de manière à personnaliser facilement le code généré pour la plateforme matérielle cible. Par conséquent, la qualité logicielle est devenue fortement corrélée aux paramètres de configuration ainsi qu'au générateur lui-même. Dans ce contexte, il est devenu indispensable de vérifier le bon comportement des générateurs.

Cette thèse établit trois contributions principales :

**Contribution I: détection automatique des inconsistances dans les familles de générateurs de code:**

Dans cette contribution, nous abordons le problème de l'oracle dans le domaine du test non-fonctionnel des générateurs de code. La disponibilité de multiples générateurs de code avec des fonctionnalités comparables (c.-à-d. familles de générateurs de code) nous permet d'appliquer l'idée du test métamorphique en définissant des oracles de test de haut-niveau (c.-à-d. relation métamorphique) pour détecter des inconsistances. Nous définissons la relation métamorphique comme la comparaison entre les variations de performance et de l'utilisation des ressources des différentes versions, générées à partir de la même famille de générateurs de code. Une inconsistance est détectée lorsque le code généré présente un comportement inattendu par rapport à toutes les implémentations équivalentes de la même famille.Nous évaluons notre approche en analysant la performance de Haxe, un langage de programmation de haut niveau impliquant un ensemble de générateurs de code multi-plateformes. Les résultats expérimentaux montrent que notre approche est capable de détecter plusieurs inconsistances qui révèlent des problèmes réels dans cette famille de générateurs de code.

**Contribution II: une approche pour l'auto-configuration des compilateurs.**

Le grand nombre d'options de compilation des compilateurs nécessite une méthode efficace pour explorer l'espace d’optimisation. Ainsi, nous appliquons, dans cette contribution, une méta-heuristique appelée Novelty Search pour l'exploration de cet espace de recherche. Cette approche aide les utilisateurs à paramétrer automatiquement les compilateurs pour une architecture matérielle cible et pour une métrique non-fonctionnelle spécifique tel que la performance et l'utilisation des ressources. Nous évaluons l'efficacité de notre approche en vérifiant les optimisations fournies par le compilateur GCC. Nos résultats expérimentaux montrent que notre approche permet d'auto-configurer les compilateurs en fonction des besoins de l'utilisateur et de construire des optimisations qui surpassent les niveaux d'optimisation standard. Nous démontrons également que notre approche peut être utilisée pour construire automatiquement des niveaux d'optimisation qui représentent des compromis optimaux entre plusieurs propriétés non-fonctionnelles telles que le temps d'exécution et la consommation des ressources.

**Contribution III: Un environnement d'exécution léger pour le test et la surveillance de la consommation des ressources des logiciels.**

Enfin, nous proposons une infrastructure basée sur les micro-services pour assurer le déploiement et la surveillance de la consommation des ressources des différentes variantes du code généré. Cette contribution traite le problème de l'hétérogénéité des plateformes logicielles et matérielles. Nous décrivons une approche qui automatise le processus de génération, compilation, et exécution du code dans le but de faciliter le test et l'auto-configuration des générateurs. Cet environnement isolé repose sur des conteneurs système, comme plateformes d'exécution, pour une surveillance et analyse fine des propriétés liées à l'utilisation des ressources (CPU et mémoire).

**Introduction en français :**

**Contexte :**

Les techniques émergentes de l’ingénierie dirigée par les modèles et de la programmation générative ont permis la création de plusieurs générateurs (générateurs de code et compilateurs). Ceux-ci sont souvent utilisés afin de faciliter le développement logiciel et automatiser le processus de génération de code à partir des spécifications abstraites. En effet, les utilisateurs sont devenus capables de synthétiser rapidement des artefacts logiciels pour une large gamme de plateformes logicielles et matérielles. En outre, les générateurs modernes comme les compilateurs C, sont devenus hautement configurables, offrant de nombreuses options de configuration à l'utilisateur de manière à personnaliser facilement le code généré pour la plateforme matérielle cible. Par conséquent, la qualité logicielle est devenue fortement corrélée aux paramètres de configuration ainsi qu'au générateur lui-même. Dans ce contexte, il est devenu indispensable de vérifier le bon comportement des générateurs. D’une part, il est important de tester les programmes générés afin de détecter des anomalies de générateurs et les corriger. D’autre part, il est crucial d’aider les utilisateurs à bien paramétrer les générateurs afin de satisfaire les exigences des plateformes logicielle et matérielle cibles.

**Motivations :**

Aujourd’hui, les générateurs populaires tels que GCC, LLVM, etc., sont fréquemment utilisés dans l’industrie afin de traduire le code source écrit en langage humain vers un code machine de bas niveau (p. ex. binaires, exécutables). Ces générateurs, connus sous le nom compilateurs, offrent une large gamme d'options de configuration aux utilisateurs pour contrôler le comportement du générateur.

Différentes catégories d'options peuvent être activées pour aider les développeurs à déboguer, optimiser l’application générée ou encore sélectionner l'architecture matérielle cible.

Par exemple, GCC version 4.8.4 offre une large sélection d'options en ligne de commande, y compris plus de 150 options d'optimisation. Cela constitue un espace de configuration très large avec 2150 combinaisons d'optimisation possibles pouvant être appliquées par l'utilisateur pour optimiser la qualité du code généré (comme le temps d’exécution, la taille du binaire, le temps de compilations, etc.). En outre, la construction d'une seule séquence d'optimisation pour tous les programmes en entrée est impossible car les interactions entre les optimisations sont trop complexes et difficiles à définir. De plus, l'impact des optimisations est fortement corrélé au matériel cible et au code source d'entrée.

Cet exemple montre combien il est difficile de paramétrer les générateurs (tel que les compilateurs) dans le but de produire un code machine qui satisfait différentes propriétés non fonctionnelles.

Il est aussi essentiel de tester les générateurs afin d’assurer que la génération automatique de code n’introduit aucune anomalie dans le produit final. Cela entrainerait une perte de confiance de la part des utilisateurs qui n’utiliseront plus ces générateurs lors du développement logiciel. En conséquence, les développeurs de générateurs doivent bien vérifier et tester le bon comportement du code automatiquement généré.

Contrairement aux compilateurs qui disposent de plusieurs solutions de test, il en existe peu pour évaluer automatiquement les comportements des générateurs de code, dont aucune ne se base sur le test des propriétés non-fonctionnelles.

En effet, les générateurs de code sont moins utilisés et expérimentés dans l'industrie par rapport aux compilateurs. Ils sont également difficiles à tester puisqu'ils impliquent un ensemble de technologies très complexes et hétérogènes.

Le test des générateurs de code implique principalement les créateurs/experts de l’outil. Néanmoins, les utilisateurs sont également responsables de cette *validation* puisqu'ils reportent continuellement les anomalies rencontrées lors de la génération de code.

De nombreuses approches ont été proposées afin de tester le comportement fonctionnel du code généré. Cependant, peu de solutions évaluent les propriétés non-fonctionnelles, à savoir les propriétés liées à la performance et l’utilisation des ressources du code généré.

En résumé, les générateurs (compilateurs et générateurs de code) sont des composants essentiel pour la production automatique du code. La qualité du logiciel généré est directement corrélée à la qualité du générateur lui-même. Tant que la qualité des générateurs est maintenue et améliorée, la qualité des artefacts logiciels générés s'améliore également puisque toute anomalie avec ces générateurs affecte directement la qualité du produit logiciel final.

En particulier, lorsque la génération automatique de code est utilisée, nous identifions deux problèmes majeurs qui menacent la qualité du logiciel généré:

D'une part, l’espace de configuration très large pose un important défi aux utilisateurs qui doivent sélectionner les meilleures options d'optimisation, répondant à certaines exigences non-fonctionnelles.

D'autre part, la complexité des générateurs de code ainsi que le manque de solutions pour évaluer les propriétés non-fonctionnelles du code généré représente un obstacle pour les créateurs de générateurs qui veulent assurer la qualité du logiciel automatiquement généré.

**Problématiques :**

Nous avons identifié trois problématiques pour le test et la configuration automatique des générateurs.

* **Le test des générateurs de code (le problème de l'oracle):**

Le test des générateurs de code relève le problème de l’oracle. Un oracle de test est le mécanisme par lequel un testeur peut déterminer si un test passe ou pas. En ce qui concerne les tests non-fonctionnels des générateurs de code, ce problème devient très récurrent parce que il est assez difficile de déterminer le comportement non-fonctionnel attendu du code à tester (p. ex. la consommation de mémoire). Déterminer si les mesures non-fonctionnelles correspondent à une anomalie de générateur ou non, n'est pas aisé. Pour contourner ce problème, des techniques telles que les tests métamorphiques sont appliquées pour tester des programmes sans définir explicitement un oracle. Cette technique emploie des relations métamorphiques de haut-niveau pour vérifier automatiquement les résultats des tests. Alors, quels types d'oracles pouvons-nous définir? Comment pouvons-nous détecter automatiquement des anomalies de générateurs? Toutes ces questions soulèvent des défis importants dans le test des générateurs de code.

* **La configuration automatique des compilateurs (exploration de l'espace de recherche des optimisations):**

La plupart des compilateurs fournissent un très grand nombre d'optimisations qui permettent d’améliorer la qualité du code généré. Cependant, pour explorer un tel espace d'optimisation, les utilisateurs doivent évaluer leurs effets selon une propriété non-fonctionnelle spécifique tel que le temps d’exécution, la taille du binaire, etc. Du fait de la difficulté de la construction d’une séquence d’optimisation donnée, de la complexité des interactions et de l'effet imprévisible des optimisations, les utilisateurs trouvent des difficultés à choisir la meilleure configuration de compilateur qui satisfait une exigence non-fonctionnelle spécifique.

* **La surveillance de la consommation des ressources du code généré (gérer la diversité des plateformes logicielles et matérielles):**

Pour évaluer les propriétés liées à l'utilisation des ressources du code généré (par des compilateurs ou des générateurs de code), les développeurs doivent compiler, déployer et exécuter les artefacts logiciels générés sur différentes plateformes d'exécution. Ensuite, ils utilisent souvent plusieurs profileurs, débogueurs et outils de surveillance spécifiques à la plateformes afin de trouver des anomalies lors de l'exécution du code généré. En raison de l'hétérogénéité des plateformes d'exécution logicielle et matérielle, la collecte d'informations sur l'utilisation des ressources du code généré devient une tâche très couteuse, puisque les développeurs doivent analyser et vérifier le code généré pour chacune des plateformes cibles à l'aide d'outils spécifiques.

Les défis de cette recherche peuvent être résumés dans les questions de recherche suivantes:

**RQ1.** Comment pouvons-nous aider les développeurs à tester automatiquement le code généré et détecter des anomalies non-fonctionnelles de générateurs de code ?

**RQ2.** Comment pouvons-nous aider les utilisateurs des compilateurs à choisir automatiquement la configuration d’optimisation adéquate qui satisfait une exigence non-fonctionnelle spécifique?

**RQ3.** Comment pouvons-nous faciliter la mise en place d’outils de surveillance de la consommation de ressources dans un environnement hétérogène?

**Contributions :**

Cette thèse établit trois contributions principales. Elles sont brièvement décrites dans le reste de cette section.

**Contribution I:** détection automatique des inconsistances dans les familles de générateurs de code.

Dans cette contribution, nous abordons le problème de l'oracle dans le domaine du test non-fonctionnel des générateurs de code. La disponibilité de multiples générateurs de code avec des fonctionnalités comparables (c.-à-d. familles de générateurs de code) nous permet d'appliquer l'idée du test métamorphique en définissant des oracles de test de haut-niveau (c.-à-d. relation métamorphique) pour détecter des inconsistances. Nous définissons la relation métamorphique comme la comparaison entre les variations de performance et de l'utilisation des ressources des différentes versions, générées à partir de la même famille de générateurs de code. Une inconsistance est détectée lorsque le code généré présente un comportement inattendu par rapport à toutes les implémentations équivalentes de la même famille.

Nous évaluons notre approche en analysant la performance de Haxe, un langage de programmation de haut niveau impliquant un ensemble de générateurs de code multi-plateformes. Nous évaluons les propriétés liées à la performance et à l'utilisation des ressources du code généré pour cinq plateformes logicielles cibles. Les résultats expérimentaux montrent que notre approche est capable de détecter plusieurs inconsistances qui révèlent des problèmes réels dans cette famille de générateurs de code.

**Contribution II:** une approche pour l'auto-configuration des compilateurs.

Comme nous l'avons indiqué précédemment, le grand nombre d'options de compilation nécessite une méthode efficace pour explorer l'espace d’optimisation. Ainsi, nous appliquons, dans cette contribution, une méta-heuristique appelée Novelty Search pour l'exploration de cet espace de recherche. Cette approche aide les utilisateurs à paramétrer automatiquement les compilateurs pour une architecture matérielle cible et pour une métrique non-fonctionnelle spécifique tel que la performance et l'utilisation des ressources. Nous évaluons l'efficacité de notre approche en vérifiant les optimisations fournies par le compilateur GCC. Nos résultats expérimentaux montrent que notre approche permet d'auto-configurer les compilateurs en fonction des besoins de l'utilisateur et de construire des optimisations qui surpassent les niveaux d'optimisation standard. Nous démontrons également que notre approche peut être utilisée pour construire automatiquement des niveaux d'optimisation qui représentent des compromis optimaux entre plusieurs propriétés non-fonctionnelles telles que le temps d'exécution et la consommation des ressources.

**Contribution III:** Un environnement d'exécution léger pour le test et la surveillance de la consommation des ressources des logiciels.

Enfin, nous proposons une infrastructure basée sur les micro-services pour assurer le déploiement et la surveillance de la consommation des ressources des différentes variantes du code généré. Cette contribution traite le problème de l'hétérogénéité des plateformes logicielles et matérielles. Nous décrivons une approche qui automatise le processus de génération, compilation, et exécution du code dans le but de faciliter le test et l'auto-configuration des générateurs. Cet environnement isolé repose sur des conteneurs système, comme plateformes d'exécution, pour une surveillance et analyse fine des propriétés liées à l'utilisation des ressources (CPU et mémoire). Cette infrastructure constitue un terrain de jeu pour tester et régler les générateurs. Cette contribution répond principalement à RQ3, mais celle-ci est particulièrement utilisée pour valider les expériences portées dans RQ1 et RQ2.