Origine et évaluation des clones

Sommaire

Les outils de recherche de similarités dans du code source s'intéressent à la récupération de zones de similarité généralement dénommées clones ou correspondances (en anglais *matches*). Il est néanmoins préalablement nécessaire de s'interroger sur une définition de clone afin de pouvoir définir plus formellement le champ d'application de ces outils.

Pour une première approche, un clone de code source est une portion de code présentant un certain degré de similitude avec une autre portion de code, ces deux portions formant une paire de clones. On définit alors un groupe de clones comme un ensemble de portions de code dont chacune des paires présente un certain degré de similarité.

Nous supposons que nous manipulons des langages de programmation impératifs présentant pour unité de base la fonction retournant un résultat (éventuellement vide) pour un paramètre communiqué (n-uplet de valeurs). Ces langages peuvent ensuite définir des éléments structurels englobant les fonctions tels que des classes ou objets, des unités de compilation ou des paquetages. Nous ne nous intéressons donc pas aux langages déclaratifs (tel que XML) ou aux langages logiques (tel que Prolog).

2.1 Taxonomie des clones

Dans cette section, nous classons les clones selon deux critères principaux. Le premier critère porte sur l'origine du clone : s'agit-il d'un clone effectif lié à une opération de copie intra-projet, inter-projets et avec quelle motivation ? Dans un second temps, nous étudions les niveaux des clones selon leur étendue sur le code source. Nous ne nous intéressons pas ici au degré d'approximation des clones qui sera analysé dans le chapitre suivant consacré aux opérations d'édition et d'obfuscation sur les clones.

2.1.1 Clones phénoménologiques

Quels sont les processus à l'origine de l'apparition de clones intra- et inter-projets ? Une première possibilité est liée à un procédé de copie de code par le développeur dont les motivations peuvent être diverses. Au sein d'un même projet, cette opération permet de réutiliser rapidement du code dans un nouveau contexte. Lorsque la copie est réalisée d'un projet extérieur, celle-ci peut être légitime ou non. Dans ce deuxième cas, il s'agit d'une opération de plagiat généralement suivie de procédés d'obfuscation afin de rendre la détection du clone moins aisée. Enfin d'autres clones de niveau structurel (architecture de paquetages ou unités de compilation) peuvent témoigner de l'utilisation de patrons de conception classiques.

Clones intra-projet

La copie de morceaux de code dans l'optique d'une réutilisation dans de nouveaux contextes est un processus (malheureusement) courant chez la plupart des programmeurs. Il apparaît ainsi selon plusieurs études [103, 102, 104] sur certains projets Open Source que leur volume pourrait être réduit jusqu'à 15% par la factorisation de morceaux de code copiés. Un moyen simple de détecter à leur source ces clones phénoménologiques pourrait être de surveiller les opérations de copie-collage de code au sein de l'environnement de développement utilisé [68]. Il serait ainsi possible d'inciter le programmeur à factoriser son code pour éviter toute redondance. Cependant, si dans certains cas la factorisation est possible, dans d'autres contextes celle-ci demeure impossible à cause des limitations du langage utilisé. Ainsi par exemple, un langage tel que Java n'offrant pas de mécanisme de généricité de type primitif nécessite la copie de code avec adaptation de types pour utiliser du code avec des types primitifs différents[1]. Dans certains cas, en programmation objet, l'intégration de plusieurs fonctionnalités déjà implantées peut se heurter à l'absence d'héritage multiple. Une autre situation problématique est rencontrée lorsque le code correspondant à une fonctionnalité est éparpillé au sein d'un projet (aspect). Une solution serait d'étendre le langage pour surmonter ces limitations (introduction du support de composition de caractéristiques [88] ainsi que des aspects) ou

[1] On pourra se reporter à de nombreux exemples du JDK notamment avec les méthodes de la classe Arrays nécessitant la réimplantation de méthodes de tri pour tous les types primitifs.

alors d'utiliser des outils de génération automatique de code (pratique couramment utilisée pour le développement de *Software Product Lines*).

La réduction du volume de code d'un projet par la factorisation de clones permet principalement une diminution de l'effort de maintenance généralement justifiée par un souci temporel et financier. La fiabilité et la sécurité du code produit est également un critère important car un bug causé par un exemplaire de code cloné a une faible probabilité d'être corrigé spontanément sur les autres exemplaires. D'autre part, une réécriture optimisée d'un exemplaire de clone peut ne pas être reportée. Dès lors le choix délibéré d'introduire une duplication de code ne peut être rationnellement motivé que par un gain de productivité à court terme qui surpasserait le gain à long terme d'un code factorisé. Ce choix est notamment réalisé lorsque la production d'un code abstrait nécessite un effort de développement important et/ou lorsque sa compréhension serait difficile. Introduire des clones peut également être utile dans des cas d'optimisation du code à l'exécution ou pour l'adaptation de composantes sur des architectures différentes (tel que par exemple le module multi-thread du serveur web Apache [124]).

Pour notre travail, nous nous intéressons principalement à la recherche de zones de code dupliqué sur un référentiel de code à un instant donné. L'étude de l'évolution de zones dupliquées au cours de la vie d'un projet [98] présente néanmoins un réel intérêt afin de juger du rôle fonctionnel des clones et de la nécessité de refactorisation. Un groupe de clones apparaissant entre deux versions peut être caractérisé, pour les versions suivantes du projet, par divers destins évolutionnistes. Des nouveaux exemplaires de code dupliqué peuvent se greffer au groupe (nouvelles opérations de copie-collage) ou au contraire être supprimés par factorisation. Des exemplaires peuvent faire l'objet de modifications homogènes ou au contraire évoluer indépendamment entre deux versions. Une évolution indépendante peut être la manifestation de corrections de problèmes non propagés par oubli à d'autres exemplaires ou alors plus légitimement le reflet d'une profonde adaptation contextuelle ; la copie initiale fournit alors un code squelette destiné à être remodelé. Une évolution indépendante n'exclut cependant pas une reconvergence ultérieure du code. L'opération de clonage de code est alors un choix de développement qui constitue une solution transitoire choisie dans l'incertitude des fonctionnalités et de l'architecture globale du projet. On notera qu'un des principaux écueils du suivi de groupes de clones entre versions reste de tracer temporellement les morceaux de code similaires.

Clones inter-projets légitimes

L'emprunt de code d'un projet vers un autre permet généralement un gain de temps de développement même si quelquefois certaines adaptations plus ou moins importantes doivent être menées. Cette pratique, pouvant consister à la réutilisation de portions plus [101] ou moins importantes de code (méthodes, classes et unité de compilation) peut être légitime si la licence du projet source du clone l'autorise. C'est le cas notamment des licences Open Source[2] autorisant les œuvres dérivées et donc l'emprunt de code. Cependant une licence Open Source est libre de fixer ses propres conditions quant à la licence de l'œuvre dérivée : ainsi si les licences de type BSD n'imposent aucune condition quant au projet destination du clone (si ce n'est de conserver les informations concernant l'auteur du code original), les licences de type GPL conditionnent la redistribution d'œuvres dérivées sous les mêmes termes. Il convient donc de choisir judicieusement la licence du projet destination. La maintenance du code copié peut

[2] Au sens de l'Open Source Initiative (`http://www.osi.org/`). Ces licences sont également qualifiées de libres.

Licence du projet destination → ↓ Licence du projet source	OS+SA	OS	!OS
OS+SA (ex : (A)GPL, CeCILL ...)	Oui	Non	Non
OS (ex : BSD, Apache, MIT, Artistic ...)	Oui	Oui	Oui
!OS	Non	Non	Non

FIG. 2.1 – Matrice de compatibilité de licences de logiciels

ensuite être menée indépendamment du projet source ou alors les nouvelles versions du projet source peuvent être utilisées avec des risques d'incompatibilité.

Dans le cadre de clones inter-projets légitimes, une adaptation du code au contexte du projet destination peut être attendue mais sans tentatives d'obfuscation des modifications. Dans la plupart des cas, la licence du projet source nécessite de laisser dans le projet destination les informations concernant la paternité du morceau copié.

Clones inter-projets illégitimes

Non respect du droit de copie La réutilisation illégitime de morceaux entre différents projets intervient lorsque les projets source et destination du clone n'ont pas le même titulaire des droits et ne disposent pas de licences compatibles autorisant la copie. Nous pouvons ainsi schématiquement classer les licences généralement utilisées pour des projets informatiques en trois grandes catégories :

1. Licences Open Source permissives (OS). Ces licences n'imposent pas de restriction sur l'utilisation du programme issu du code, imposent la disponibilité du code source sous une forme lisible par un humain avec la possibilité de redistribuer des copies exactes ou dérivées sans restriction de licence sur ses œuvres dérivées.
2. Licences Open Source avec redistribution sous des conditions identiques (OS+SA — *Share Alike*). La redistribution d'œuvres dérivées du projet nécessite d'utiliser une licence avec des conditions identiques de liberté d'utilisation, de disponibilité du code source et de redistribution d'œuvres dérivées.
3. Licences non Open Source (!OS). Elles sont définies négativement comme licences ne satisfaisant pas les libertés d'utilisation, de disponibilité du source et de redistribution des licences Open Source. Ces licences sont généralement qualifiées de privatrices ou propriétaires. Par défaut, un projet ne disposant pas de licence explicite entre dans cette catégorie.

Nous présentons en figure 2.1 une matrice schématisant les cas de légitimité de copie de code selon les types de licence des projets source et destination. Des cas de copie illégitime peuvent survenir entre projets Open Source lorsque le projet source requiert une redistribution sous conditions identiques et pas le projet destination. Des réutilisations illégitimes de code peuvent avoir pour cadre l'échange de code entre projets non-OpenSource et projets Open Source. Concernant l'emprunt de code OS+SA par des projets !OS, l'interrogation de bases indexant des projets Open Source pourrait mettre en évidence ce phénomène. Certaines sociétés [139, 138] se sont spécialisées dans la mise à disposition d'un tel service pour des éditeurs de logiciels commerciaux.

```
1 int c1, c2; int tmp;
  ...
  tmp = c1; c1 = c2; c2 = tmp;
```

(a) Échange de variables *int*

```
1 char a, b; char temp;
  ...
  tmp = a; a = b; b = temp;
```

(b) Échange de variables *char*

FIG. 2.2 – Deux exemplaire de code d'échange de variables en C

Plagiat Un autre cas étranger aux problématiques de droits de copie concerne les cas de plagiat intervenant typiquement au sein de projets réalisés par des étudiants dans le cadre de leurs études. Si une copie volontaire de code entre plusieurs projets apparaît illégitime, il est également possible de noter des clones involontaires liés à la finalité fonctionnelle similaire des projets soumis. Il est également possible d'observer des cas de copie avec des projets extérieurs (projets préexistants ou projet réalisé par un tiers sur demande de l'étudiant) qui peuvent plus ou moins être tolérés en fonction de la taille du code emprunté et du bon signalement de l'auteur original de l'œuvre. Certaines mesures préventives peuvent toutefois être employées pour limiter les cas de code copié en proposant par exemple plusieurs sujets de projet inédits pour lesquels la réutilisation de code existant ainsi que la copie entre étudiants de la même promotion sera difficile.

Obfuscation Dans le contexte de clones illégitimes, il est fortement probable que le plagiaire cherche à rendre la détection de l'emprunt du code plus difficile. Différentes méthodes d'obfuscation peuvent alors être utilisées dont certaines seront examinées dans le chapitre 4. Dans le cadre de notre travail, nous cherchons à permettre la détection de clones en présence d'utilisation de méthodes d'obfuscation en limitant le report de faux-positifs.

2.1.2 Niveaux des clones

Clones locaux

Un clone local concerne une portion de code similaire de taille modeste à l'échelle d'une fonction. La spécificité d'un clone local réside dans le fait que ses sous-éléments (typiquement des instructions) ne sont en règle générale pas commutatifs. Il peut néanmoins être possible de déterminer des dépendances entre instructions afin de segmenter le corps d'une fonction en différents groupements d'instructions indépendants. Nous présentons en figure 2.2 un exemple de deux exemplaires de clones concernant l'échange de valeurs de deux variables avec types différents et renommage d'identificateurs de variable. Ce clone pourrait être refactorisé par l'écriture d'une macro ou d'une fonction prenant pour paramètres les adresses des variables et leur taille mémoire.

Les outils de détection de clones étudiés s'intéressent dans leur majorité à la détection de clones locaux avec une tolérance plus ou moins importante quant aux opérations d'édition entre clones.

Clones structurels

Un clone structurel est un clone de taille plus étendue qu'un clone local : un tel clone peut concerner à différentes échelles plusieurs groupes d'instructions indépendantes d'une fonction,

un ensemble de fonctions, une classe, une unité de compilation, un paquetage voire un projet entier. Ces clones étant de taille importante, ils comportent généralement de nombreuses opérations d'édition. Les sous-éléments d'un groupe de clones structurels peuvent être dans un ordre différent selon les exemplaires. En effet, si l'ordre d'instructions présente une importance à l'échelle d'une fonction, les membres d'une classe ou d'une unité de compilation (variables membres, méthodes...) sont généralement commutatifs. Nous proposons au chapitre 11 une méthode permettant l'agrégation de clones locaux proches afin de créer des clones structurels approximatifs plus étendus : un exemple d'un tel clone (au niveau fonctionnel) révélé par cette méthode est spécifié en figure 2.3.

Clones creux

Contrairement aux clones structurels qui peuvent être décomposés comme une juxtaposition de clones locaux de localisations plus ou moins proches, les clones creux ne présentent comme similarité que la macro-structure du code source. Ainsi, par exemple, deux classes présentant des méthodes de signature équivalente (mêmes types de paramètres) sans que le corps de ces méthodes soit similaire peuvent être considérées comme des clones structurels creux. De même, deux portions de code présentant la même hiérarchie de structures de contrôle mais avec des instructions différentes peuvent être reportées comme des clones locaux creux.

Localiser les clones creux peut être utile dans une optique de réingénierie afin de réorganiser la hiérarchie des classes par la création de nouveaux ancêtres communs lorsque ceci est nécessaire. Ces clones structurels permettent de mettre en lumière des schémas de conception spécifiques et peuvent également permettre de classifier des unités de source de domaines fonctionnels proches mais d'implantations différentes.

La représentation la plus adaptée à l'étude des clones creux est l'arbre de syntaxe du code : on pourra alors les mettre en évidence par la recherche de sous-arbres homomorphes jusqu'à une certaine profondeur ou avec abstraction de certains nœuds.

2.2 Évaluation des clones détectés

Un clone détecté est un clone dont l'existence est découverte par l'utilisation d'un système de détection de clone, qu'il soit automatique ou humain. On remarque que les méthodes algorithmiques nécessitent l'utilisation de paramètres afin d'ajuster le report de clones conduisant à l'obtention de groupes de clones potentiellement différents. Quant à un juge humain, selon certains critères définis (degré de similarité, utilité de la factorisation, ...), il peut rechercher des clones présents au sein de projets avec une part inévitable de subjectivité.

Nous proposons maintenant quelques définitions pouvant servir de base à une approche quantitative et qualitative de mesure d'efficience d'un outil de recherche de clones. Il s'agit d'introduire les deux dimensions que sont l'extensibilité et la peuplabilité d'un groupe de clones et de les relier à la notion de pertinence.

```
72 public class Javac12 extends DefaultCompilerAdapter {
       public boolean execute() throws BuildException {
           attributes.log("Using␣classic␣compiler", Project.MSG_VERBOSE);
75         Commandline cmd = setupJavacCommand();
           OutputStream logstr = new LogOutputStream(attributes, Project.MSG_WARN);
           try {
               // Create an instance of the compiler, redirecting output to
               // the project log
80             Class c = Class.forName("sun.tools.javac.Main");
               Constructor cons = c.getConstructor(new Class[] { OutputStream.class, String.class });
               Object compiler = cons.newInstance(new Object[] { logstr, "javac" });
               // Call the compile() method
               Method compile = c.getMethod("compile", new Class [] { String[].class });
85             Boolean ok = (Boolean)compile.invoke(compiler, new Object[] {cmd.getArguments()});
               return ok.booleanValue(); }
           catch (ClassNotFoundException ex) {
               throw new BuildException("Cannot␣use␣classic␣compiler,␣as␣it␣is␣not␣available"+
                 "␣A␣common␣solution␣is␣to␣set␣the␣environment␣variable"+
90               "␣JAVA_HOME␣to␣your␣jdk␣directory.", location); }
           catch (Exception ex) {
               if (ex instanceof BuildException) {
                   throw (BuildException) ex;
               } else {
95                 throw new BuildException("Error␣starting␣classic␣compiler:␣", ex, location); }
           } finally {
               try {
                   logstr.close();
               } catch (IOException e) {
100                // plain impossible
                   throw new BuildException(e); } } } }
```

(a) ant.taskdefs.compilers.Javac12

```
67 public class KaffeRmic extends DefaultRmicAdapter {
       public boolean execute() throws BuildException {
           getRmic().log("Using␣Kaffe␣rmic", Project.MSG_VERBOSE);
70         Commandline cmd = setupRmicCommand();
           try {
               Class c = Class.forName("kaffe.rmi.rmic.RMIC");
               Constructor cons = c.getConstructor(new Class[] { String[].class });
               Object rmic = cons.newInstance(new Object[] { cmd.getArguments() });
75             Method doRmic = c.getMethod("run", null);
               String str[] = cmd.getArguments();
               Boolean ok = (Boolean)doRmic.invoke(rmic, null);
               return ok.booleanValue();
           } catch (ClassNotFoundException ex) {
80             throw new BuildException("Cannot␣use␣Kaffe␣rmic,␣as␣it␣is␣not␣available"+
                "␣A␣common␣solution␣is␣to␣set␣the␣environment␣variable"+
                "␣JAVA_HOME␣or␣CLASSPATH.", getRmic().getLocation() ); }
           catch (Exception ex) {
               if (ex instanceof BuildException) {
85                 throw (BuildException) ex;
               } else {
                   throw new BuildException("Error␣starting␣Kaffe␣rmic:␣", ex, getRmic().getLocation()); } } } }
```

(b) ant.taskdefs.rmic.KaffeRmic

FIG. 2.3 – Clone structurel intra-projet au niveau fonctionnel présent dans Eclipse-Ant (mis en évidence par consolidation de correspondances en 11.5)

2.2.1 Pertinence d'un groupe de clones

Définition 2.1. Un groupe de clones $c_{1..n} = (c_1, c_2, \cdots, c_n)$ est dit pertinent selon un oracle ψ ssi $\psi(c_{1..n}) =$ vrai. Un oracle d'évaluation de pertinence ψ' est dit plus restrictif que l'oracle ψ ssi pour tout $c_{1..n}$, $\psi'(c_{1..n}) \implies \psi(c_{1..n})$.

L'oracle ψ se présente en pratique généralement sous les traits d'un évaluateur humain qui n'est donc pas déterministe. Une autre alternative, libérée de toute subjectivité humaine, serait d'utiliser une méthode algorithmique déterministe pour juger de la pertinence des clones. Il paraît néanmoins difficile d'établir des critères d'évaluation algorithmique objectifs de clones.

Des oracles d'évaluation de pertinence algorithmiques peuvent être fournis par des outils de recherche automatique de similarité : un groupe de clones est alors pertinent ssi il est mis en évidence par l'outil. Nous définissons ainsi l'oracle ψ_R évaluant un groupe de clones $c_{1..n}$ comme pertinent ssi les exemplaires de clones $c_1, \cdots, c_n$ sont égaux deux à deux selon une représentation R choisie. Il est alors possible de proposer des méthodes algorithmiques pour implanter ψ_R où R peut être une forme lexemisée (recherche de groupes de facteurs exacts sur structures d'indexation de suffixes tel que vu au chapitre 6) ou une représentation par arbres de syntaxe (recherche de sous-arbres égaux par hachage, cf chapitre 10). Afin de limiter le report de clones trop peu volumineux, on pourra après avoir défini une fonction de volume sur clone poser un oracle $\psi_R^{\geq t}$ plus restrictif que l'oracle précédent, ajoutant comme condition à la détermination de pertinence un volume de clone plus grand ou égal à t.

Une opération d'évaluation humaine est quant à elle temporellement très coûteuse et peut présenter l'inconvénient d'être psychologiquement ennuyeuse sur un ensemble de groupes de clones conséquent. Ainsi, lorsque nous cherchons à évaluer la pertinence moyenne d'un ensemble de groupes de clones, nous pouvons ne sélectionner qu'un échantillon statistiquement significatif de groupes qui seront soumis à un oracle humain, en supposant le résultat extrapolable à l'ensemble des clones. Une technique d'échantillonnage a notamment été employée par Bellon [86] pour l'évaluation de différents outils de recherche de clones. [100] présente une expérience d'évaluation de pertinence d'une vingtaine de clones trouvés par un outil automatique de recherche, CCFinder [71], sur le projet Open Source de moteur de base de données PostgreSQL par huit juges humains : moins de la moitié des clones reportés faisaient l'objet d'une évaluation de pertinence (ou non-pertinence) consensuelle partagées au moins par sept juges.

2.2.2 Extensibilité et réductibilité des groupes de clones

Rechercher des similitudes exactes selon une représentation R (implantation de l'oracle ψ_R) présente des limitations lorsque des opérations d'édition sont attendues entre exemplaires de clones. Il peut donc être utile de s'interroger sur la pertinence de l'extension de clones à partir de germes que sont des clones exacts. À cet effet, nous introduisons la notion d'extensibilité et de réductibilité sur un groupe de clones. Intuitivement, il s'agit de déterminer si des morceaux de code participant à un groupe de clones peuvent être étendus tout en gardant le caractère pertinent de ce groupe ou au contraire réduits, si le groupe est non pertinent.

Définition 2.2. Deux morceaux de code c et d s'intersectent ssi c et d partagent un sous-morceau de code commun. c est inclus dans d ssi c et d s'intersectent avec pour morceau

de code commun c. Par extension deux groupes de morceaux de code $C = (c_1, c_2, \cdots, c_n)$ et $C' = (c'_1, c'_2, \cdots, c'_n)$ s'intersectent (resp. C est inclus dans D) ssi pour chaque $i \in [1..n]$, il existe j tel que c_i intersecte c'_j (resp. c_i est inclus dans c'_j). Étant donné un ensemble de groupes de morceaux de code, un morceau de code c est dit *propre* si celui-ci n'est inclus dans aucun autre morceau de code extrait de l'ensemble des groupes.

Ainsi par exemple, les morceaux de code *abc* et *bcd* s'intersectent par *bc* alors que *bcd* est inclus dans *abcde*. Si l'on considère ces quatre morceaux comme représentants de groupes de clones, seul *abcde* est un morceau propre.

Définition 2.3. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ pertinent est extensible ssi il est possible de trouver un groupe de clones pertinents $(c'_1, c'_2, \cdots, c'_n)$ tels que c_1 soit inclus dans c'_1, c_2 soit inclus dans c'_2, ..., c_n soit inclus dans c'_n. Dans le cas contraire, le groupe de clones est dit maximal.

Définition 2.4. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ non pertinent est réductible ssi il est possible de trouver des clones pertinents $(c'_1, c'_2, \cdots, c'_n)$ tels que c'_1 soit inclus dans c_1, c'_2 soit inclus dans c_2, ..., c'_n soit inclus dans c_n.

Il est donc souhaitable de proposer un outil de recherche de clones proposant des groupes pertinents maximaux.

2.2.3 Peuplabilité et dépeuplabilité des groupes de clones

Si les méthodes de recherche de clones approximatifs utilisant des techniques de consolidation s'intéressent uniquement à l'obtention de groupes de clones de cardinalité 2, des méthodes exactes peuvent construire directement des groupes de clones de cardinalité k quelconque sans faire usage d'étapes de regroupement séparées telles qu'envisagées au chapitre 13. Dans cette optique, il peut être intéressant de se demander s'il serait possible d'ajouter un clone à un groupe sans qu'il perde son caractère pertinent ou au contraire si un groupe non pertinent pourrait devenir pertinent grâce à la suppression d'un clone.

Définition 2.5. Un groupe de clones $(c_1, c_2, \cdots, c_n)$ pertinent est peuplable ssi il existe un morceau de code c_{n+1} tel que $(c_1, c_2, \cdots, c_n, c_{n+1})$ soit également un groupe pertinent. Un groupe pertinent non-peuplable est dit complet.

Définition 2.6. Un groupe de clones $c_{1..n} = (c_1, c_2, \cdots, c_n)$ non pertinent est dépeuplable ssi il existe un sous-groupe de $c_{1..n}$ pertinent.

Un groupe peut être peuplable soit par intégration d'un morceau de code n'appartenant à aucun autre groupe, soit par intégration d'un sous-groupe ou d'un groupe entier externe. On peut remarquer que si deux groupes pertinents $c_{1..k}$ et $c'_{1..k'}$ possèdent deux morceaux c_i et $c_{i'}$ tel que $(c_i, c_{i'})$ soit un groupe pertinent, cela ne constitue pas une condition suffisante à la fusion de $c_{1..k}$ et $c_{1..k'}$ en un seul groupe pertinent, la relation de clone n'étant pas toujours transitive. Nous pouvons également noter qu'un même morceau peut appartenir à plusieurs groupes pertinents.

La figure 2.4 présente un groupe de trois morceaux de code lexemisés soit extensibles sur la droite de deux lexèmes, soit peuplable par l'ajout d'un nouveau morceau. Les deux groupes constitués par extension et peuplement peuvent coexister en tant que résultat d'une méthode de recherche de clones. Les trois morceaux du groupe étendu sont qualifiables de propres tandis que le groupe peuplé comporte comme unique morceau propre `int d = a`.

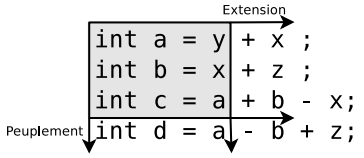

FIG. 2.4 – Peuplement et extension de groupes (représentation lexemisée avec abstraction des identificateurs)

2.2.4 Précision et rappel

Définition 2.7. La précision du résultat (ensemble de groupes de clones reporté) $\mathcal{C}$ notée $\mathcal{A}(\mathcal{C})$ est définie comme le ratio du volume global des clones pertinents détectés sur le volume global des clones détectés.

Définition 2.8. Le rappel du résultat $\mathcal{C}$ noté $\mathcal{R}(\mathcal{C})$ est défini comme le volume global des clones pertinents détectés sur le volume global des clones pertinents.

La difficulté de la détermination de la précision et du rappel d'un résultat est liée à l'obtention de l'ensemble des groupes de clones pertinents. Comme vu précédemment, la pertinence n'étant pas une donnée absolue, un oracle est utilisé qui pourra être un évaluateur humain, un autre outil de recherche voire un procédé de vote par une combinaison de méthodes. Lorsque le coût d'évaluation de pertinence est trop élevé, la précision et le rappel peuvent être statistiquement estimés. La précision d'un sous-ensemble significatif de l'ensemble reporté peut être calculée est extrapolée à l'ensemble total.

En ce qui concerne le rappel, nous pouvons prendre pour base de calcul un sous-ensemble de groupes de clones pertinents. Ainsi, le comparatif de Bellon [86] évalue le rappel en introduisant artificiellement des clones (avec différents degrés d'édition) au sein de projets qui sont ainsi considérés comme pertinents et en déterminant le nombre de clones détectés parmi ces clones. Cette approche permet ainsi de déterminer plusieurs mesures de rappel selon l'oracle d'évaluation de pertinence utilisé qui pourra se limiter à des clones exacts ou à des clones présentant un degré d'édition plus ou moins important. Une tentative de classification des clones est présentée dans le chapitre 4 selon les opérations d'édition qui les différencient.

Les définitions de précision et de rappel introduisent la notion de volume global de clones qui peut intuitivement être résumé comme une mesure de la quantité d'information portée par l'ensemble des morceaux de code participant à des groupes de clones. Cette mesure peut par exemple être réalisée en découpant la représentation du code en atomes élémentaires, chacun associé à un volume, le volume d'un morceau étant la somme des volumes des atomes le constituant.

2.2.5 Pertinence floue

Une limitation des définitions proposées réside dans l'absence de prise en compte de *pertinence floue*. En effet, un groupe de clones non pertinent pouvant être transformé en groupe pertinent par écourtement ou dépeuplement minime peut dégrever la précision et le rappel. En revanche, aussi bien la précision que le rappel sont insensibles à l'existence de groupes pertinents qui pourraient être étendus ou peuplés.

Ainsi, sur la chaîne $u = abcabdab$, le report des groupes de clones $ab : \{(u[1..2], u[4..5], u[7..8])\}$ (groupe pertinent non-prolongeable et non-extensible selon un oracle considérant uniquement pour pertinents des facteurs exacts) ou des groupes $ab : \{(u[1..2], u[4..5]\}$ et $ab : \{(u[4..5], u[7..8]\}$ (groupes peuplables), ou $a : \{u[1], u[4], u[7]\}$ et $b : \{u[2], u[5], u[8]\}$ (groupes extensibles) engendrent la même précision et le même rappel (ici de 1) car tous les lexèmes appartenant à des clones pertinents, et uniquement ceux-ci sont couverts.

2.2.6 Quantification de la similarité d'exemplaires de clones

Définition 2.9. Une métrique de similarité s sur l'espace des exemplaires de clones est une fonction associant à chaque paire de morceaux de code un vecteur reflétant son degré de similarité.

Les morceaux de code d'un groupe de clones ne sont généralement pas des copies exactes de code : des opérations d'édition plus ou moins importantes peuvent être notées, que ce soit pour l'adaptation du clone à un nouveau contexte ou alors pour masquer une copie illégitime. Il est intéressant d'établir une métrique de similarité entre plusieurs morceaux. Celle-ci peut être de nature vectorielle et concerner plusieurs critères : similarité des noms de variables, distance d'édition du code... En règle générale, le vecteur de similarité entre deux morceaux de code est calculé en utilisant la représentation intermédiaire de code spécifique à l'outil de détection employé. Deux familles de représentations seront plus particulièrement étudiées dans ce document : les chaînes de lexèmes ainsi que les arbres de syntaxe.

Exemples de métriques de similarité

Une approche simple employable lorsque le code source est représenté par des chaînes de lexèmes est de calculer une métrique de similarité basée sur la distance d'édition (ou distance de Levenshtein [14]) entre les deux séquences (ce qui est réalisé par l'utilisation d'une technique d'alignement telle que décrite au chapitre 5). Au préalable cela nécessite de définir une fonction de volume associée aux éléments manipulés dans la représentation intermédiaire (ici des séquences de lexèmes). Nous discuterons plus en détail des métriques de similarité au chapitre 12.

Pour une représentation par arbre de syntaxe, il est possible de définir également une distance d'édition basée sur des opérations élémentaires (ajout/suppression de nœud, transformation de nœud) pour transformer un sous-arbre en un autre. Nous évoquons les techniques de programmation dynamique afin de calculer une telle distance en section 5.5. Calculer une distance d'édition entre deux sous-arbres ne paraît judicieux que si ces deux structures présentent préalablement un certain niveau de similarité. Ainsi lorsque nous nous limitons à des

opérations de transformation sur certains types de nœuds ou de sous-arbres, nous pouvons utiliser des méthodes de hachage de sous-arbres selon différents profils d'abstraction comme abordé dans le chapitre 10. Deux sous-arbres identiques selon un profil d'abstraction général peuvent alors être examinés plus en détails avec des profils plus spécialisés pour en déduire une distance d'édition.

* *
*

Nous avons pu au cours de ce chapitre examiner les causes et motivations sous-jacentes à l'apparition de code dupliqué au sein d'un même projet ou dans des projets différents. Cette duplication peut apparaître à l'échelle locale voire à une échelle macroscopique. À l'issue de l'obtention de portions de code dupliquées par un outil spécifique, une des principales difficultés concerne l'évaluation de la pertinence de ces résultats, notion toujours relative au jugement humain ou à la comparaison avec d'autres outils automatiques de recherche. Au-delà d'une simple fonction booléenne, la notion de pertinence devrait également intégrer la possibilité d'étendre, de réduire, de peupler ou de dépeupler un groupe de clones. L'objectif d'une méthode algorithmique de recherche de similarité consiste alors à proposer l'ensemble exhaustif des groupes de clones complets, non-extensibles et pertinents selon le jugement de la majorité de ses utilisateurs humains. Cet objectif peut être atteint lorsque la pertinence est équivalente à l'égalité exacte d'une représentation sous-jacente par chaîne de lexèmes ; nous utilisons à cet effet des structures d'indexation de suffixes dans le chapitre 6. Toutefois lorsque des similarités approchées sont recherchées, la relation de duplication liant les paires d'exemplaires d'un groupe de clones n'est plus transitive. Nous obtenons alors des groupes de deux exemplaires (2-correspondances) pour lesquels la notion de peuplabilité ne fait plus sens : nous ne nous intéressons alors qu'à l'extensibilité. Nous aborderons ainsi au chapitre 11 une méthode de consolidation de clones exacts sur arbre de syntaxe pour laquelle nous nous questionnerons au sujet de la réductibilité et de l'extensibilité des 2-correspondances trouvées. Ce n'est que dans un second temps que nous nous intéresserons à la possibilité de fusionner des groupes de 2-correspondances proches au sein de groupes plus peuplés au chapitre 13.