Supporting Code Clone Inspection using Parameterized Clone Pattern

Udo Borkowski
abego Software GmbH, Aachen
Germany
ub@abego-software.de

Abstract

Code clone inspection is an integral part of software clone management to assess the quality of clones or the tools reporting them, to decide how to resolve code clone issues, and so on. As clone inspection is a manual process its feasibility is limited especially when working with large numbers of clones. This is rather critical as clone detection tools may return many clones even when applied to medium sized projects. Parameterized Clone Pattern abstracts clones using parameters and merge them into shared "patterns". The more abstract view on clones reduces the number of necessary manual clone inspections. In addition this approach can be used to provide intuitive visualizations of clones, report potential problems, suggest possible next steps etc.. This makes Parameterized Clone Pattern a good candidate to improve clone inspection efficiency.

Introduction

Code inspection is a rather time consuming task. Bellon [1] mentioned it took him 40 sec in average to check a clone pair when working on the „Bellon Clone Detection Benchmark" [2]. Considering the benchmark contains more than 300.000 clone pairs it would have taken 2 person years to inspect all of them. For this reason Bellon only inspected 2 % of them.

There seem to be two options to improve this situation: reduce the time per clone pair inspection or reduce the number of clone pairs to inspect manually. The concept of PCP introduced in this paper uses both options.

Parameterized Clone Pattern

Parameterized Clone Pattern (PCP, or shorter „clone pattern") is an attempt to make Ira Baxter's definition of code clones as „segments of code that are similar according to some definition of similarity" more concrete.

Definition: Parameterized Clone Pattern

A PCP is a function $P(p_1, ..., p_n)$ [n >= 0] returning a code fragment. p_i is called a parameter of P.

Definition: Clone Class

A clone c_i is part of the clone class P iff there is a PCP $P(p_1, ..., p_n)$ and arguments $a_1,...a_n$ such that c_i is equal to $P(a_1, ..., a_n)$. The definition of „equal" depends on the context. E.g. whitespaces or comments may be ignored when comparing.

Assuming two things are similar if they have something in common but may vary in certain aspects a PCP defines the commonality between clones while the parameters provide a way to express the variability. Applying the PCP with specific arguments produces instances of the PCP (i.e. code fragments or „clones").

This definition of PCP covers all types of clones [3]:

- exact clones are parameterless PCPs,
- for parameter-substituted clones the parameters correspond directly to identifiers or literals being substituted.
- for structure-substituted clones the parameters match the text of the subtrees being replaces, and
- for non-contiguous clones extra parameters are introduced to represent the „gaps".

Find Clone Pattern using Clone Detection

Code clone detection can be used to find new PCPs in an incremental, interactive way.

We assume a clone detection tool returns its result as a set $Clones = \{cl_1, ..., cl_k\}$ with each cl_i („clone list") representing a list of code fragments. If $|cl_i| = 2$ the clone list is called a clone pair. All elements in cl_i are considered to be clones to each other. Therefore it should be possible to find a corresponding PCP. For a given clone list `cloneList` the following algorithm can be used to find its clone pattern(s).

Algorithm: findClonePattern

```
result = Ø
for each c in cloneList
  m = null
  if (result ≠ Ø)
     m = match(c, result)
  if (m == null)
     m = FixedTextPattern(c)
  if (result ≠ Ø)
     gcp = generalize(result, m)
     if (gcp ≠ null)
        in result replace specific
                 patterns by more general
        continue;
  add m.clonePattern to result;
```

`match(...)` tries to match a given code fragment c to a set of clone patterns and returns the most specific match as a tupel (clonePattern, a_1, ..., a_n), if possible. clonePattern(a_1, ..., a_n) will return text equal to c.

`generalize(...)` finds a generalized pattern that covers m.clonePattern and some patterns from result (the specific ones). The generalized pattern keeps track of the clone pattern more specific to it.

The implementation of `match` and `generalize` de-

pend on the representation used for the Clone Pattern. E.g. for string templates with placeholders for parameter substitution `generalize` may use sequence alignment [4] to align patterns. The “inserts” and “deletes” become new parameters.

The result of findClonePattern is then merged with the list of already known patterns. This may also involve some generalisation steps.

Manual verification is suggested if

- the result contains elements not yet in the know patterns, e.g. to classify the clone pattern („accept“, „reject“) or to check if the elements are not „too general“. In this situation one could locate inconsistent substitutions (see chapter “Clone Inspection and Clone Pattern”).
- the result contains more than one element. E.g. one would check if the generalize function did not recognize a possible generalisation, or the clone list (typically provided by a clone detection tool) contained clones that are too different from each other.

Clone Inspection and Clone Pattern

PCPs can support clone inspection in various ways.

- They provide a way to classify clones (e.g. „accept“, „reject“) by annotating clone patterns with appropriate information. This information is then applicable to all clones in the PCP's clone class. The classification only needs to be done once for the PCP and not for every individual clone.
- PCPs help detecting inconsistently substituted clones. This case can often easily be recognized by checking the arguments of the clones of one PCP: for a given parameter all values are equal except for some „outliers“. The pattern may be annotated to issue a warning if this parameter does not have the required values. Typically this situation is initially recognized during the „findClonePattern“ phase when a new pattern is created.
- PCPs can detect inconsistent changes to clones between subsequent versions of the source code. To achieve this it is necessary to recall the clones with their PCPs and arguments of the previous version. For details see [5].
- PCPs reduce the number of clones that have to be analyzed manually. This is mainly due to the fact that many operations (like „accept“, „reject“) can be done on a PCP level, covering multiple clones at once.
- PCPs simplify manual inspection by making the variability more obvious. E.g. this can be achieved by presenting tables with the arguments for each clone. Also one may use the PCP to visualize the parameters in the actual source code, e.g. when comparing two clones.

Results

The presented approach was implemented in the tool CloneInspector, which is currently capable of processing Java code. It was initially evaluated on the dataset provided by the „Bellon Clone Detection Benchmark“.

The first results using the Baxter clone pairs for the eclipse-ant project are quite promising (see Table 1). After introducing 12 clone patterns the number of unmatched clones was more than halved and the number of clone pairs to check was reduced to nearly a tenth. Only 5 of these pattern were application specific. The other patterns are candidates for reuse on other projects as they represent common Java idioms.

	a)	b)	c)	d)	e)
1. no patterns	-	-	1086	9686	
2. patterns to halve remaining clones	**12**	**5**	540	974	15%
3. general pat. to halve remain. clones	**17**	**0**	541	1250	0%
4. adding 10 extra general pat. to 2.	**22**	**5**	458	646	13%
a) total pattern count b) app-specific pattern count c) unmatched clones d) remaining clone pairs e) % of clones matched by app-specific pattern					

Table 1: Applying Clone Patterns on Baxter's clones of eclipse-ant

More evaluation is necessary to verify these results. Especially the clone detection approach may influence the efficiency of the PCP approach.

Related Work

PSPs are strongly related to the work by Brenda Baker on parameterized pattern matching [6], especially as the current implementation of CloneInspector is mainly based on string substitution. However PCPs are able to cover a wider range of variability that may be exploited in future work.

Summary

The concept of Parameterized Clone Pattern (PCP) was introduced and a constructive approach to find them using clone detection tools was presented. The applicability of PCPs in clone inspection was shown, suggesting PCPs could improve the efficiency of the manual code clone inspection.

References

[1] Bellon, Stefan: *Vergleich von Techniken zur Erkennung duplizierten Quellcodes*. Diplomarbeit, Universität Stuttgart, 2002

[2] Website "Detection of Software Clones", http://www.bauhaus-stuttgart.de/clones/

[3] Koschke, Rainer: *Frontiers of software clone management.* 24th IEEE International Conference on Software Maintenance, 2008, 119–128

[4] Gusfield, Dan: *Algorithms on Strings, Trees and Sequences*. Cambridge University Press 1999

[5] Borkowski, Udo: *C4D oder Wie ich lernte, mit Code Clones zu leben.* 2004 http://www.udo-borkowski.de/C4D/ 2004.

[6] Baker, Brenda S.: *A theory of parameterized pattern matching: algorithms and applications.* Symposium on Theory of Computing 1993, 71-80