Annotation Support for Generic Patches

Georg Dotzler, Ronald Veldema, Michael Philippsen
University of Erlangen-Nuremberg, Computer Science Department
Programming Systems Group, Erlangen, Germany
georg.dotzler@cs.fau.de, veldema@cs.fau.de, philippsen@cs.fau.de

***Abstract*—In large projects parallelization of existing programs or refactoring of source code is time consuming as well as error-prone and would benefit from tool support. However, existing automatic transformation systems are not extensively used because they either require tedious definitions of source code transformations or they lack general adaptability. In our approach, a programmer changes code inside a project, resulting in before and after source code versions. The difference (the generated transformation) is stored in a database. When presented with some arbitrary code, our tool mines the database to determine which of the generalized transformations possibly apply. Our system is different from a pure compiler based (semantics preserving) approach as we only suggest code modifications.**

Our contribution is a set of generalizing annotations that we have found by analyzing recurring patterns in open source projects. We show the usability of our system and the annotations by finding matches and applying generated transformations in real-world applications.

***Keywords*-programming tools; optimizations; patches; code-refactoring;**

I. Introduction

When developers fix bugs, apply API changes or parallelize code, they often perform similar code changes multiple times which causes programming overhead. For example, Alice parallelizes a loop in a module of her project and guards some critical data structures against race conditions in this loop. Bob encounters a similar loop in one of his modules. If he has no information about Alice's prior work, he also needs to identify the same critical data structures.

To avoid the duplicate work, a tool that applies Alice's transformation to Bob's code and presents this suggestion to him is helpful. The tool could extract the transformation by comparing Alice's initial loop with the improved loop. But the transformation extracted from Alice's single example lacks generality. For example, assume that in Alice's code a variable is incremented. What if in Bob's code the variable is decremented? Without generalization, finding a loop that exactly fits the transformation from Alice is unlikely. A known solution to this issue is to analyze multiple slightly different examples. However, one cannot expect Alice to provide multiple suitable variations of her loop.

In our system, we keep the example-based approach and use annotations for generalization that Alice adds as comments to her code. The example-based approach frees Alice from the task of specifying AST (abstract syntax tree) transformations explicitly. For example, it is not necessary to define an edit operation that inserts a 'parallel-loop' node in the AST at the appropriate place. Besides generalizations, the annotations can also be used to constrain the application of transformations. For example, a useful constraint for a parallel loop is a minimal trip count as the parallelization of small loops is not beneficial. To avoid a complex annotation language we allow our annotations to refer to plugins that can test additional code properties.

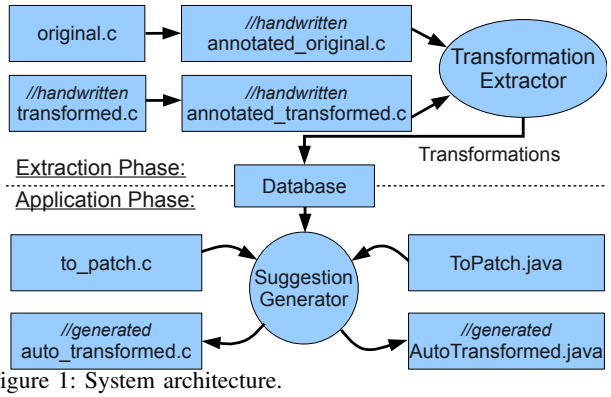

Figure 1: System architecture.

As our system does not guarantee to only generate correct suggestions, Bob must determine if a suggested transformation is applicable and useful for his code. Therefore, Alice is not required to fulfill the near impossible task of specifying exactly under which conditions her transformation is applicable. Furthermore, it allows our system to even make suggestions whose preconditions cannot be proven formally at compile-time, but may still hold according to Bob's domain know-how.

Many applications are programmed in a set of related languages (C, C++, Java, etc.). As these languages are similar, a patch created by Alice in Java can often still be applied to a C++ program. Of course, this is only possible if the patch does not use language specific features or libraries that are missing on the target platform.

Our contributions are (1) identification of annotations to generalize example-based transformations and to avoid over-generalization by means of constraints, (2) extraction of cross-language reusable transformations from comparing two annotated example programs, and (3) a usability study of our suggestion-based code-transformation system.

```
//# match(original, 1) {
  for (int i=0; i<size; i++)
    c[i] = a[i+1] + b[i];
//# }

//# match(original, 2) {
  for (int i=0; i<size; i++)
    c[i] = a[i+1] - i;
//# }
```

Figure 2: Original side of two transformations.

```
//# match(transformed, 1) {
  #pragma omp parallel for
  for (int i=0; i<size; i++)
    c[i] = a[i+1] + b[i];
//# }

//# match(transformed, 2) {
  #pragma omp parallel for
  for (int i=0; i<size; i++)
    c[i] = a[i+1] - i;
//# }
```

Figure 3: Parallelized code of Fig. 2.

```
for (int j=0; j<size; j++)
  c[j] = a[j+1] + max(b[j], c[j]);
```

Figure 4: The Transformation extracted by comparing the code parts in Figs. 2 and Fig. 3 does not match here.

II. Annotations

We have identified six types of annotations that are required to both generalize transformations and to specify constraints. Alice who wants to place a transformation in the database annotates both her before and after source codes. Our tool extracts the transformation from these annotated source files and places the transformation into a database (see Fig. 1). The annotations are only required in the Extraction Phase. Bob does not have to annotate his code.

The basic syntax for our annotations is:

```
//# <annotation-name>(args)?
    <plugin: <name>(arg, arg, ..)>*
```

The "//#" prefix separates normal comments from our annotations. Then follow the annotation's name, optional arguments, and zero or more calls to program analysis functions, implemented as plugins. The number of required plugins depends on the annotation in question. Annotations can precede a subsequent code or code block, which we call the associated statement. As a plugin often needs to refer to a specific AST fragment of the annotation's associated statement, we supply a special pattern clause (to be used as plugin call argument):

`pattern([occurrence,]? expr [, name]?)`

Occurrence and name are optional arguments. Given `pattern(2, c[i], #a)` and an associated statement `c[i]=c[i]+1`, the pattern clause evaluates to a reference to the second `c[i]` and assigns the name "#a" to that expression. We will later show the pattern clause in use.

A. Match Annotation

Alice first needs to mark each single transformation using the `match` annotation. See Fig. 2 (annotated original code) and Fig. 3 (annotated patched code). Here, Alice did two unrelated edits in a single file and instructs our tool to generate two transformations and to place them in the database. The match annotation's first argument tells which is the original code and which is the transformed code. The second argument is a unique ID that links the match block in the original code to the transformed block. Here, the tool creates two transformations that match all `for` loops with the specified structure. While the default transformation extractor already generates transformations that abstract from variable names, it expects a `for`-range starting from a constant and running up to some integer variable. Furthermore, the upper transformation expects a body with a single assignment of the form: $\texttt{array}_1\texttt{[loopvar] = array}_2\texttt{[loopvar + constant] + array}_3\texttt{[loopvar]}$. Although our system supports the commutative law the transformation is still far too specific and cannot be applied to the code shown in Fig. 4. In this case the `wildcard` annotation described below is needed.

B. Wildcard Annotation

The `wildcard` annotation broadens the applicability of a transformation. Each generalizable term in Alice's code must be identified by a `pattern` argument so that it can be passed to a plugin that tests the pattern's properties.

Fig. 5 adds the wildcard annotation to the first loop of Fig. 2. With `plugin:effectFree(pattern(1, b[i])` any side-effect free expression can replace `b[i]` in the sum, in particular the `max(b[j], c[j])` expression that is present in Fig. 4.

A wildcard's plugin limits the amount of code the wildcards abstracts. Here, the `effectFree()` plugin limits the number of consumed AST nodes to a single term and only then tests that the expression is indeed effect-free.

C. Use Annotations

The wildcard annotation only consumes code, but does not insert the matched code in the transformed code as this is not always wanted. To insert the matched expression, Alice first names the matching pattern. In Fig. 5 the clause `pattern(1, b[i], #x)` assigns the name `#x` to the matching code. In the transformed code in Fig. 6 the `use` annotation specifies where to insert the named code (replaces `b[i]` in the result). Hence with both the `wildcard` and the `use` annotations, the transformation generated from Fig. 5 $\rightarrow$ Fig. 6 can be applied to the loop of Fig. 4 and produces the desired result.

D. Constraint Annotation

A system with only `wildcard` annotations generates too many suggestions, e.g. even a loop with few iterations could be suggested to be parallelized. The `constraint` annotation allows Alice to narrow the applicability of the transformation.

```
//# match(original, 1) {
  for (int i=0; i<size; i++) {
    //# wildcard
        plugin:effectFree(
        pattern(1, b[i], #x))
     c[i] = a[i+1] + b[i];
  }
//# }
```

Figure 5: Wildcard annotation for `b[i]`.

```
//# match(transformed, 1) {
  #pragma omp parallel for
  for (int i=0; i<size; i++) {
    //# use(
        #x == pattern(1, b[i]))
      c[i] = a[i+1] + b[i];
  }
//# }
```

Figure 6: Parallelized code of Fig. 5, `b[i]` is replaced by `#x`.

```
//# match(original, 1) {
//# constraint
      plugin:checkCondition(
          pattern(size), ">", 1000)
  for (int i=0; i<size; i++)
      c[i] = a[i+1] + b[i];
//# }
```

Figure 7: Constraint added to Fig. 2.

Fig. 7 constraints the transformation to loops with more than 1000 iterations. The `checkCondition` plugin must prove that the `size` variable is larger than 1000. A failed constraint does not cause our tool to discard the suggestion, it is only burdened by a penalty score. This penalty score is used to rank the suggestions in the list presented to Bob. Besides the constraint annotation two other factors influence the penalty score. The more nodes are matched by a wildcard annotation the higher is the penalty score. The more nodes are used in the transformation the smaller is the penalty score as larger transformations are often more useful.

E. Insert Annotation

Some transformations require code to be inserted outside of the regions marked by the `match` annotation. This can be done by the `insert` annotation. For instance, let us assume that OpenMP requires an initial `openmp_init()` (e.g. in `main()`). Fig. 8 shows how Alice can do this. The argument '1' of the `insert` links this insert-block to the match-block with the same ID. The plugin `fpos` returns the location in the to-be-transformed program. The code is inserted at the `begin` of `main` using the name and signature of `main`. Other available placement plugins for the `insert` annotation can return all loops, all `return` statements, etc. The second plugin `stmtExists` tests if `openmp_init` is already called. This avoids repeated initializations.

F. Generator Annotation

In the above examples we always explicitly referred to OpenMP for the parallelization. But what if Bob wanted to use CUDA instead? We therefore need some way to emit different statements that are based on Bob's preferences. This is were the `generator` annotation comes in.

In the transformed example code of Fig. 9, the plugin `createParFor()` is responsible for generating the code for the chosen parallelization method. For example, if the generated source code is C, the plugin adds `#pragma omp parallel` to the function.

G. Plugin System

Annotation descriptions often invoke plugins. The existing plugin library has four categories: code analysis, consumption, placement, and transformation plugins.

Alice can choose from the code analysis plugins to express constraints about types and values, for example, `effectFree()` in Fig. 5 and `checkCondition()` in Fig. 7. In this category there are also plugins to analyze the call-tree. For example, the `isPar()` plugin used in the STAMP benchmark (see Sec. III-B) determines whether the found pattern is inside a parallel region or not. Moreover, there are plugins for live variable analysis, dominance analysis, type analysis, etc. Another example is the `isShared` plugin (Fig. 10) that determines whether an expression reads or writes memory that is shared by multiple threads.

Consumption plugins restrict the number of statements or expressions that are consumed by a wildcard. For example, the `pE` plugin (used in the Apache case study in Sec. III-A) matches only primary expressions (e.g. identifiers) and excludes arithmetics. To match all expressions there is the `matchAllExpr` plugin (Fig. 10). The `matchAllStmts` plugin matches entire blocks of statements.

The placement plugins are used by `insert`. One example is the `fpos` plugin in Fig. 8.

Transformation plugins affect the resulting code of the suggestion and are used in generator annotations. In Fig. 9 the generator is responsible for adding the parallel code parts to the suggestion.

```
//# match(transformed, 1) {
  #pragma omp parallel for
  for (int i=0; i<size; i++)
    c[i] = a[i+1] + b[i];
//# }
//# insert(1,
  plugin:fpos("begin","main"),
  plugin:stmtExists("false")) {
    openmp_init();
//# }
```

Figure 8: Initialization Example.

```
//# match(transformed, 1) {
//# generator plugin:createParFor()
  for (int i = 0; i < size; i++)
      c[i] = a[i+1] + b[i];
//# }
```

Figure 9: Generator for parallel code.

```
//# match(original, 1) {
//# constraint plugin:isPar()
//# wildcard
     plugin:isShared(
       pattern(c[i][j], #Z)),
     plugin:matchAllExpr(
       pattern(value, #R))
     c[i][j] = value;
//# }
```

Figure 10: Annotated original code.

The main goal of the plugin system is to make Alice's job easy. If Alice cannot find a suitable plugin, there are two options. First, Alice can always use a more generic plugin at the penalty of Bob seeing more suggestions. Second, Alice (or plugin expert Carol) can spend some time on writing a new plugin if she is familiar with our framework.

III. Evaluation

We use three case studies to examine our tool. We first apply a patch to the Apache HTTP Server (http://httpd.apache.org/) to explore how our system copes with a larger project. The second benchmark examines how well our tool aids in parallelizing an application. This involves locating code points that access shared data which requires more complex plugins, and stresses our plugin system. Finally, the third case study examines cross-language capabilities by applying a generic patch extracted from a sequential and multi-threaded Java program to a C++ program with OpenMP.

A. Apache's Httpd

We study a code change that occurred between the versions 2.2.5 and 2.2.18. In version 2.2.5, programmers used two calls to clone a buffer. First they allocated sufficient memory (`apr_palloc`) and then they copied the data to the new buffer with `memcpy`. In the newer version they used a single call to `apr_pmemdup`.

Between the two versions this change has been (manually) applied 23 times. As similar code is written in slightly different ways in all the spots (using different variables and types of data to clone), copy-and-paste does not work.

In contrast, our tool only requires one annotated example with four `wildcards` in the source and four `use` annotations in the transformed fragment.

With this input, it finds and suggests all 22 other transformations in the around 200 Apache source files (134,978 lines of code) of the old version. The search and suggestion generation takes 16s (8s for I/O+parsing, 6s for match searching, and 2s for misc.).

B. STAMP Benchmark Suite

STAMP [1] is a suite of nine benchmarks for transactional memory research. For this case study we stripped the atomic regions from the original benchmark code, but left the code inside the atomic regions in the source files. The program analyis plugins are exercised by two transformations that recreate atomic regions.

The first transformation re-introduces atomic blocks to each assignment to shared variables. Besides a simple assignment (`c[i][j] = value;`) the source side of the transformation uses three wildcards with three different plugins (see Fig. 10). The first two plugins test if the access occurs in a parallel context (`isPar()`) and if the data access `isShared`. The `matchAllExpr()` plugin consumes all possible right-hand-sides of the assignment.

As this transformation is designed to create atomic blocks for every single assignment, its application to the unannotated and stripped version of the STAMP benchmarks resulted in 305 suggestions. The total time needed to create the transformations averages to 2.9s. The suggestion generation takes 1.3s per file.

The second transformation merges adjacent atomic blocks to increase granularity. When applying this transformation to the stripped versions of the STAMP benchmarks, our tool generates 122 suggestions. This demonstrates that our tool is helpful in parallelizing applications. In combination the two transformations generate many atomic blocks that are similar to the blocks found in the original benchmarks.

C. SOR

To demonstrate the usability of our tool's cross-language and cross-framework abilities we choose an implementation of the Successive Over-Relaxation method (SOR) from the Java Grande benchmark suite [2] which is a typical numeric stencil application. The extracted transformation is then applied to a sequential stencil kernel in C, taken from the Splash [3] benchmark's `jacobi` function. In this case, only a simple `match` annotation is needed in the original code. In the transformed code, a generator is used that automatically patches the `for` loop to either OpenMP code or threaded code depending on available libraries. The transformation extracted from the annotated Java SOR code can also be applied to generate a C-OpenMP version.

IV. Related Work

Two works are closely related to ours. Robbes and Lanza [4] propose a system that first records the changes that a programmer makes and then uses the record to create a transformation template. Also similar to us, Meng et al. [5] use an example-based approach, that compares before and after sources and is based on the ChangeDistiller by Fluri et al. [6]. Compared to both approaches, our system is more flexible due to our plugin system. Robbes and Lanza rely on the AST representation of the source code to use all features whereas our annotations are directly linked to the written source code. The second approach by Meng et al. [5] does not include any wildcard support.

The system from Tate et al. [7] also learns transformations by examining the differences between two programs. To generalize their transformations, they use a proof checker to detect places in the code that are structurally different but compute the exact same results. Relying on a proof checker for these tasks restricts their system to only provably correct transformations and does not allow the system to learn patterns that are correct under certain run-time conditions or according to Bob's domain know-how. Our tool overcomes this issue by relying on programmer insight.

Instead of letting the system extract program transformations from examples, some tools require the manual definition of the transformation. For example in Stratego/XT [8] one has to describe the AST operations directly. As shown by Visser [9], an improvement to these tools is the use of the concrete syntax of the target language of the transformations. Manniesing et al. [10] propose a transformation language to transform `for` loops to vectorized loops. Another DSL for expression rewrite rules is presented by Baxter et al. [11] where the transformation tool is a part of a larger toolset.

Padioleau et al. [12] use semantic patches, based on parameterized textual patches generated by standard Unix diff/patch tools, to define general transformations. The transformation approach with semantic patches has been extended by Andersen and Lawall [13]. Instead of having the programmer annotate the code, their system 'learns' the semantic patch annotations by examining a set of diffs (source code repository commits) and by finding commonality between them. However, this approach requires a large set of diffs of similar changes as input.

Other systems target API changes, for example, [14]. They do so by automatically mining either framework or application changes. These systems cannot capture arbitrary changes, however.

V. Conclusion

Based on a before and after approach, it is low effort to express a code transformation for a specific case. To constrain and generalize such an example transformation we introduced source code annotations. These annotations are simple enough to avoid a steep learning curve, but powerful enough to express complex patterns as they can use the underlying plugin system. Our test cases show that our tool allows beautifications, synchronization, patches and cross-language parallelism to be added to programs. By not restricting ourselves to semantics-preserving transformations, we give a large set of useful suggestions compared to other tools.

With our approach it might also be possible to use a transformation database that is filled by crowd-sourced techniques. Future work will show how large communities can add optimizations, fixes, etc. to central databases even if some contributors do not have enough background knowledge to prove the correctness of their patches.

VI. Acknowledgements

This project has been supported in part by the Embedded Systems Institute (ESI) and the ESI-Anwendungszentrum, http://esi-anwendungszentrum.de/.

References

[1] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC'08: Proc. IEEE Intl. Symp. Workload Characterization*, Seattle, WA, Sep. 2008, pp. 35–46.

[2] L. A. Smith, M. J. Bull, and J. Obdrzálek, "A Parallel Java Grande Benchmark Suite," in *Proc. ACM/IEEE Conf. Supercomputing*, Denver, CO, Nov. 2001, pp. 8–17.

[3] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in *Proc. 22nd Intl. Symp. Computer Architecture*, Santa Margherita Ligure, Italy, June 1995, pp. 24–36.

[4] R. Robbes and M. Lanza, "Example-Based Program Transformation," in *MoDELS'08: Proc. Intl. Conf. Model Driven Eng. Languages and Systems*, Toulouse, France, Sep. 2008, pp. 174–188.

[5] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: generating program transformations from an example," in *PLDI'11: Proc. 32nd Conf. Progr. Lang. Design and Impl.*, San Jose, CA, June 2011, pp. 329–342.

[6] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 725–743, Nov. 2007.

[7] R. Tate, M. Stepp, and S. Lerner, "Generating compiler optimizations from proofs," in *POPL'10: Proc. Symp. Principles of Progr. Languages*, Madrid, Spain, Jan. 2010, pp. 389–402.

[8] E. Visser, "Stratego: A Language for Program Transformation based on Rewriting Strategies. System Description of Stratego 0.5," in *RTA'01: Rewriting Techniques and Applications*, ser. Lecture Notes in Computer Science, vol. 2051, Utrecht, The Netherlands, May 2001, pp. 357–361.

[9] E. Visser, "Meta-Programming with Concrete Object Syntax," in *GPCE'02: Generative Programming and Component Engineering*, ser. Lecture Notes in Computer Science, vol. 2487, Pittsburgh, PA, Oct. 2002, pp. 299–315.

[10] R. Manniesing, I. Karkowski, and H. Corporaal, "Automatic SIMD Parallelization of Embedded Applications Based on Pattern Recognition," in *Proc. Intl. Europar Conf.*, ser. Lecture Notes in Computer Science, vol. 1900, Munich, Germany, Aug. 2000, pp. 349–356.

[11] I. D. Baxter, C. Pidgeon, and M. Mehlich, "DMS: Program Transformations for Practical Scalable Software Evolution," in *ICSE'04: Proc. 26th Intl. Conf. Software Eng.*, Edinburgh, Scotland, May 2004, pp. 625–634.

[12] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," *Oper. Syst. Rev.*, vol. 42, no. 4, pp. 247–260, 2008.

[13] J. Andersen and J. L. Lawall, "Generic patch inference," in *ASE'08: Proc. Intl. Conf. Automated Softw. Eng.*, L'Aquila, Italy, Sep. 2008, pp. 337–346.

[14] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proc. Intl. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*, Reno/Tahoe, NV, 2010, pp. 302–321.