Automated Migration of Build Scripts using Dynamic Analysis and Search-Based Refactoring

Milos Gligoric[1], Wolfram Schulte[2], Chandra Prasad[2], Danny van Velzen[2], Iman Narasamdya[2], and Benjamin Livshits[3]

University of Illinois at Urbana-Champaign[1], Microsoft[2], Microsoft Research[3]

gliga@illinois.edu, {schulte, chandrap, dannyvv, iman.narasamdya, livshits}@microsoft.com

Abstract

The efficiency of a build system is an important factor for developer productivity. As a result, developer teams have been increasingly adopting new build systems that allow higher build parallelization. However, migrating the existing legacy build scripts to new build systems is a tedious and error-prone process. Unfortunately, there is insufficient support for automated migration of build scripts, making the migration more problematic.

We propose the first *dynamic* approach for automated migration of build scripts to new build systems. Our approach works in two phases. First, from a set of execution traces, we *synthesize* build scripts that accurately capture the intent of the original build. The synthesized build scripts are typically long and hard to maintain. Second, we apply *refactorings* that raise the abstraction level of the synthesized scripts (e.g., introduce functions for similar fragments). As different refactoring sequences may lead to different build scripts, we use a search-based approach that *explores* various sequences to identify the best (e.g., shortest) build script. We optimize search-based refactoring with *partial-order reduction* to faster explore refactoring sequences. We implemented the proposed two-phase migration approach in a tool called Metamorphosis that has been recently used at Microsoft.

1. Introduction

At the heart of modern software development processes lies a *build system* (e.g., Ant [1], Make [7], Maven [8], MSBuild [6], and NMake [10]). The role of the build system varies widely, from compiling source code (such as C, C++, Java, or C#) into machine code or bytecode to running regression tests to launching static analysis tools to packaging and deploying applications. The actual build steps are specified in a *build script* (e.g., `Makefile`, `build.xml`, and `pom.xml`).

Being able to build software quickly and reliably is of utmost importance in today's world of never-ending requirement changes [17]. The efficiency of the build system is therefore an important enabler of developer productivity. As a result, developer teams are increasingly adopting cloud-based build systems [2, 4, 12, 17, 52]. For example, Microsoft recently developed a cloud-based build system called CloudMake [24][1]. A key feature of CloudMake is to parallelize build based on statically computed *dependencies* among tool invocations (such as `csc`, `cl`, and `powershell`).

Success of a new build system depends on the effort that developers make to migrate from existing build systems to the new system. Manual migrations are tedious and error-prone due to the size and complexity of industrial build scripts [16, 17, 47, 48]. For example, Linux developers spent a year on a failed (manual) migration[2] for version 2.5 and another year on a successful (manual) migration [17]. Automated migrations are non-trivial because accurately inferring dependencies between a multitude of tools invoked by build scripts is challenging [25, 51, 52, 62]. Existing automated approaches [5, 8] use *static analysis* and therefore only work in a limited number of cases, such as when new build scripts need not specify the dependencies explicitly, which limit parallelization. Further, static-analysis based approaches are tightly coupled to the original build system.

OOPSLA '14, October 20–24, 2014, Portland, OR, USA.

http://dx.doi.org/10.1145/2660193.2660239

[1] Although we focus on migration to cloud-based build systems, inspired by Microsoft's effort to migrate to CloudMake, there are numerous reasons for migrating from one build system to another [14, 16, 17, 47, 48].

[2] Migration from kbuild 2.4 to kbuild 2.5.

This paper introduces the first approach based on *dynamic analysis* for migrating legacy build scripts to new build systems. The dynamic nature enables migration of build scripts written for *any* existing build system to a new build system. In addition, our dynamic approach captures all dependencies necessary for parallel builds. The new approach works in two phases. In the first phase, from a set of execution traces obtained by running the old build script, we *synthesize* a new build script that accurately captures the intent of the original build. However, as is the case with runtime execution traces, these scripts contain duplicate or similar fragments as well as unstructured tool invocations. In the second phase, we apply *refactorings* that automatically raise the abstraction level of the synthesized scripts by introducing functions for similar fragments.

One can apply various refactorings on the synthesized scripts. As some refactorings can disable/enable other refactorings, different *refactoring sequences* can result in different build scripts. We use search-based refactoring [55, 61] to explore various refactoring sequences and to identify the best build script (according to a fitness function). We propose a *partial-order reduction* (POR), adopted from model checking [34, 41], as an optimization to our search-based refactoring. Our experimental evaluation shows that POR greatly reduces the exploration space.

This paper makes the following contributions:

- ⋆ **Migration Approach:** We describe the first approach that uses dynamic analysis to automatically migrate legacy build scripts to a new build system.
- ⋆ **Synthesis:** We synthesize new build scripts from captured execution traces. Because of its dynamic nature, our approach is independent of the original build system and translates entire scripts by considering multiple executions (for various build flavors).
- ⋆ **Efficient Search-Based Refactoring:** We propose an optimization of search-based refactoring: partial-order reduction. This optimization is important for faster exploration when numerous refactorings are available.
- ⋆ **CloudMake Refactorings:** We propose and formally define a set of 17 refactorings specific to CloudMake, whose goal is to raise the abstraction of the synthesized scripts. To the best of our knowledge, this is the most extensive set of refactorings for a build language.
- ⋆ **Evaluation:** We implemented our two-phase migration approach and all 17 refactorings for CloudMake in a tool called Metamorphosis. We present an evaluation of Metamorphosis on four large Microsoft projects, some containing over 1,000 legacy build scripts. The results show that Metamorphosis can migrate the build scripts of such large projects to CloudMake; that search-based refactoring reduces script size up to 46% over synthesized build scripts; and that our optimized search explores refactoring sequences up to 6.3× faster than naïve exploration.

```
Program     ::=  {Stmt}
Stmt        ::=  VarStmt | ReturnStmt
VarStmt     ::=  var identifier = Exp;
ReturnStmt  ::=  return Exp;
Exp         ::=  Lit | identifier | Exp ? Exp : Exp
                 | PrefixOp Exp | Exp InfixOp Exp
                 | Exp Invocation | Exp Refinement | LambdaExp
Lit         ::=  true | false | undef | number | string | path
                 |ObjLit | ArrLit
ObjLit      ::=  {[identifier:Exp{,identifier:Exp}]}
ArrLit      ::=  [Exp{,Exp}]
PrefixOp    ::=  - | !
InfixOp     ::=  * | + | - | >= | ...
Invocation  ::=  ([Exp{,Exp}])
Refinement  ::=  .identifier | [Exp]
LambdaExp   ::=  (identifier | ([identifier{,identifier}]) => Exp
```

Figure 1: CloudMake's grammar, in Extended BNF

2. CloudMake by Example

This section briefly introduces build systems in general, Microsoft's CloudMake [24] build system in particular, and describes migration of a simple `Makefile` to CloudMake script.

Build systems: Build systems execute build scripts to achieve various tasks including compilation, running tests, deployment, etc. Build scripts specify the tools to be executed and the dependencies between these tools. Cloud-based build systems work best when the underlying parallelism in the build script is exposed by making dependencies explicit. Existing build languages such as NMake and MSBuild, however, do not adequately expose dependencies to enable fine-grained parallelism. For instance, build systems often use build phases and not explicit dependencies to specify that all header files have to be generated before the system can be compiled.

CloudMake: CloudMake [24] is a cloud-based build system (and language) developed by Microsoft. Syntactically, CloudMake is a *purely functional* subset of TypeScript. Figure 1 shows CloudMake's grammar. Similar to other modern build systems (e.g., Google build system [4], Vesta [14], Buck [2], SCons [12], Ninja [11], and others, *fine-grained dependencies* between the tools are made explicit in CloudMake, to enable parallelization, incremental builds, and other key features of reliable high-performance build systems. Specifically, a build script has to define which files, registry keys, or environment variables each tool reads, writes, or updates. These dependencies are then used not only to create parallel schedules, but also for runtime enforcement. The runtime system ensures that

```
1  /// <reference path="Latex.ms" />
2  var name = "oopsla14.tex";
3  var bib = "oopsla14.bib";
4  var common = ["abstract.tex", "intro.tex", ...];
5  var pdflatex1 = pdflatex(name, common);
6  var bibtex2 = bibtex(bib, pdflatex1.aux);
7  var pdflatex3 = pdflatex(name, common ++
8                           bibtex2.bbl ++ bibtex2.blg);
9  var pdflatex4 = pdflatex(name, common ++
10                          bibtex2.bbl ++ pdflatex3.aux);
11 var main = copy(pdflatex4.pdf, "final.pdf");
```

(a) Refactored CloudMake script

```
// Latex.ms
function pdflatex(src, deps) {
  var result = exec({
    dir: ".", tool: "pdflatex.exe", args: [src],
    deps: src + deps, env: [],
    out: [changeExt(src, "aux"), changeExt(src, "pdf")]});
  return {aux: result.out[0], pdf: result.out[1]};}

// Common.ms
function copy(src, dst) {
  var result = exec({
    dir: ".", tool: "cp.exe", args: [src, dst],
    deps: [src], env: [], out: [dst]});
  return result.out[0];}
```

(b) Two library functions

Figure 2: Examples of library functions and refactored CloudMake script for building this paper. ("++" concatenates two arrays or appends an element to an array)

build tools behave as specified, i.e., read only what they claim and write at most what they declare.

CloudMake's extensibility is provided by the primitive exec({dir, tool, tool_args, deps, env, out}). This primitive allows for external tool invocation, including compilers and linkers, during a build. The function exec takes the following as arguments: working directory (dir); the tool to invoke (tool); a list of arguments for the tool itself (tool_args); a list of (paths to) artifacts that the tool reads (deps); environment variables that the tool may use (env); and a list of artifacts that the tool produces (out). If exec terminates successfully, it produces the artifacts whose paths are specified in out, and returns those paths.

Motivating example: We illustrate the CloudMake language using a script for building this paper, shown in Figure 2a. The build script first declares three variables (name, bib, and common) that keep the name of the main latex file, name of the bibtex file, and the list of common dependencies, respectively. Second, the script invokes four library functions. Each invocation corresponds to a step for building a pdf from latex sources. Finally, we copy the resulting pdf file to final.pdf.

Note that each function invocation (except the first) takes as input the output of the previous step, thus establishing build dependencies between the invocations (i.e., which tool takes the output of which tool). For simplicity, we assume that evaluating a CloudMake program involves evaluating variable main; the evaluation in CloudMake is demand-driven.

```
1  /// <reference path="Common.ms" />
2  var pdflatex1 = exec({
3    dir: "c:\...\oopsla14",
4    tool: "c:\program ...\pdflatex.exe",
5    args: ["oopsla14.tex"],
6    deps: ["c:\...\miktex ...\epsfig.sty", ...],
7    env: [{"PATH", "c:\program ...", ...}],
8    out: ["c:\...\oopsla14\oopsla14.aux", "c:\...\oopsla14\oopsla14.
         pdf"]});
9
10 var bibtex2 = exec({
11   dir: "c:\...\oopsla14",
12   tool: "c:\program ...\bibtex.exe",
13   args: ["oopsla14"],
14   deps: [pdflatex1.out[0], "c:\...\oopsla14\oopsla14.bib",...],
15   env: [{"PATH", "c:\program ...", ...}],
16   out: ["c:\...\oopsla14\oopsla14.blg", "c:\...\oopsla14\oopsla14.
         bbl"]});
17
18 var pdflatex3 = exec({
19   dir: "c:\...\oopsla14",
20   tool: "c:\program ...\pdflatex.exe",
21   args: ["oopsla14.tex"],
22   deps: ["c:\...\miktex ...\epsfig.sty",
23          bibtex2.out[0], bibtex2.out[1], ...],
24   env: [{"PATH", "c:\program ...", ...}],
25   out: ["c:\...\oopsla14\oopsla14.aux", "c:\...\oopsla14\oopsla14.
         pdf"]});
26
27 var pdflatex4 = exec({
28   dir: "c:\...\oopsla14",
29   tool: "c:\program ...\pdflatex.exe",
30   args: ["oopsla14.tex"],
31   deps: ["c:\...\miktex ...\epsfig.sty",
32          bibtex2.out[1], pdflatex3.out[0], ...],
33   env: [{"PATH", "c:\program ...", ...}],
34   out: ["c:\...\oopsla14\oopsla14.aux", "c:\...\oopsla14\oopsla14.
         pdf"]});
35
36 var copy5 = exec({
37   dir: "c:\...\oopsla14",
38   tool: "c:\windows ...\cp.exe",
39   args: [pdflatex4.out[1], "c:\...\oopsla14\final.pdf"]
40   deps: [pdflatex4.out[1]],
41   env: [{"PATH", "c:\program ...", ...}],
42   out : ["c:\...\oopsla14\final.pdf"]});
```

Figure 3: CloudMake script for our running example after the synthesis phase of Metamorphosis

The script in Figure 2a contains a reference to Latex.ms (the notation ///... is an include statement) that exports the definitions of pdflatex and bibtex functions. Underneath the surface, each function invokes exec, used to read and/or write the external state. In Figure 2b we show the implementation of pdflatex and a standard library function for copying a file (copy); the implementation of bibtex is similar. changeExt is a helper function that changes the extension of a file name. Note that the pdflatex function returns an object as the result with paths to aux and pdf files.

Migration to CloudMake: We briefly discuss our two-phase migration approach by migrating Makefile (shown below), originally used to build this paper, to the CloudMake script shown in Figure 2a. (Note that our work is not concerned with the complexity of the

build scripts/systems, but rather with the migration from an existing to a new build system. Therefore, we do not compare complexity of the build scripts.)

```
main:
  pdflatex oopsla14.tex
  bibtex oopsla14
  pdflatex oopsla14.tex
  pdflatex oopsla14.tex
  copy oopsla14.pdf final.pdf
```

First, we monitor the execution of `make` and synthesize a script that has one `exec` per tool invocation. The result of this phase is shown in Figure 3. In general, a script synthesized from an execution trace is long, repetitive, and difficult to maintain, since all abstractions that were present in the original build scripts have been lost.

Second, we apply a set of refactorings to raise the level of abstraction of the synthesized scripts. Our refactorings are based on our knowledge of the abstractions that CloudMake developers commonly use in practice. Developers use (single-assignment) variables for common sub-expressions such as in builds to set shared paths or flags, functions to define shared parametrized behavior, function composition to define composed functionality (e.g., call `bibtex` passing in the result of `pdflatex`), loops to summarize repetitive work (e.g., copy several files to a directory), case distinctions to specialize behavior (e.g., one flag for one build flavor and another flag for the other flavor), and modules or/and separate files to support multi-directory builds.

The challenge of automatic refactoring is to perform sophisticated pattern-matching at the level of the CloudMake abstract syntax tree (AST) and to convert them to higher-level abstractions. To illustrate some of the general principles, here we highlight several relevant refactorings applied to the script in Figure 3 to obtain the CloudMake script in Figure 2a.

- ⋆ **Properties of CloudMake's semantics**. It is common that a tool is invoked multiple times while some arguments remain the same. In our example, all invocations of the `pdflatex` have all library dependencies in common. By applying the `ExtractTOOLDependencies` refactoring, common dependency arrays (lines 6, 14, 22, and 31) would be replaced by a shared variable, line 4 in Figure 2a. This refactoring is similar to clone detection [19, 22, 40, 63].
- ⋆ **Reuse library functions**. Many tools (e.g., `csc`, `pdflatex`, etc.) are reused across projects. If an auxiliary library function is available for a given tool, we can apply the `ReuseTOOL` refactoring to identify `exec`s that can be replaced by a function invocation. This refactoring would extract the necessary arguments, insert the appropriate function invocation, and adjust the usage of the output of the function invocation. To obtain the script in Figure 2a, we perform `ReusePdflatex` (lines 2, 18, and 27), `ReuseBibtex` (line 10), and `ReuseCopy` (line 36).
- ⋆ **Introduce tools**. In cases when a tool (e.g., `csc`) is invoked a considerable number of times and no library function is available, the `IntroduceTools` refactoring can automatically create a function with common arguments and introduce the necessary function invocations. In our example, running the `IntroduceTools` refactoring would identify multiple invocation of `pdflatex` (lines 2, 18, and 27) and introduce a function with common arguments.

To obtain the best script, based on some fitness function, we use search-based refactoring [39, 54, 55, 57, 61] that explores various refactoring sequences. We discuss the *exploration phase* in Section 5.

3. Phase One: Synthesis

In this section, we describe the first phase of our migration approach. Figure 4 (left part) illustrates this phase. The inputs to this phase are build scripts written for *any* existing build system and commands that specify how to build all possible flavors (e.g., Debug and Release). Outputs are synthesized new build scripts; one build script per build flavor.

We execute two steps to obtain the synthesized build scripts. First, we execute the original build for *each build flavor* and capture the execution traces for these builds [16]. An execution trace is essentially a list of tool invocations observed during the original build. In the execution trace each tool invocation is associated with working directory, set of environment variables at the time of the invocation, arguments passed to the tool, and all input/output activities performed by the tool (e.g., file creation, file read, etc.). Metamorphosis takes an execution trace as input, analyzes the list of tool invocations, and builds a dependency graph based on the input/output information. Using the information from the dependency graph and other collected information, Metamorphosis creates one `exec` per tool invocation. Second, we synthesize a "flat" CloudMake script for each execution trace. An example of a "flat" CloudMake script is shown in Figure 3.

The key insight that underlies Metamorphosis is that, *independently* of the original build system that the software project uses, be it Make, NMake, MSBuild, Ant, Maven, or a set of loosely-connected Perl scripts, we can execute a build and monitor its actual behavior through operating system-based instrumentation. During migration the original build should be executed as "clean build" to ensure that all dependencies are captured. Note that we use existing operating system-based instrumentation with minor changes; the novelty lies in

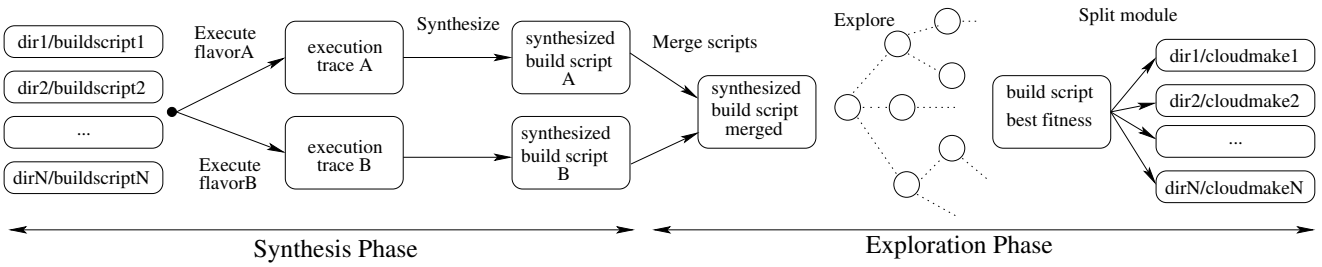

Figure 4: Two-phase migration approach using dynamic analysis and search-based refactoring

the idea of using the infrastructure to migrate build scripts from one system to another.

Process monitoring: To produce execution traces, we use Detours [42] to intercept operating system functions that perform file reads, file writes, and process creations. Detours is a library for instrumenting Win32 functions by rewriting the in-memory code for target functions. Detours also preserves the original target functions as subroutines that can be used by the instrumentation. We rewrite the functions that perform file reads and file writes to also dump information about the files being read or written by the corresponding process. We also rewrite the function that creates processes to apply the same rewriting for file reads and file writes to the created process. Thus, we are able to capture the file reads and writes performed by the child processes. (Some build scripts start only a single process, but this process spawns a great number of child processes for tool invocations.) We assume that inter-process communication between tool invocations is only by means of reading from and writing to files. For example, we do not capture communication through registry keys.

4. Refactorings in Metamorphosis

In this section we formally define a set of transformations for build scripts (Section 4.1). We use these transformations to implement 17 refactorings for CloudMake (Section 4.2).

4.1 Transformations

Notation: We express transformations as rewrite rules over script schemes in the following form

$$\mathsf{Name} :: \mathit{input} \longrightarrow \mathit{output}, \textbf{if } \mathit{precondition}$$

where *input* and *output* are CloudMake script schemes, i.e., terms that may contain free variables. A formal definition of calculus that we use is provided elsewhere [20]. Our rewrite rules are derived from (1) CloudMake's semantics; (2) data type-specific insights such as properties of paths; (3) tool-specific insights, e.g., special properties for `copy` or `csc`.

Intuitively, if a term matches the *input* and the *precondition* holds, the instantiated *input* term can be replaced by the instantiated *output* term. In the following, we refer to the *precondition* only in the text. To replace the free occurrence of an identifier x by another script part t in schema S, we write $S[t \textbf{ for } x]$. For simplicity, we assume that during substitution, bound variables are renamed to resolve name clashes. The rules below use the following naming conventions: S stands for (global) statement sequences; E for expressions; F, G, and H for functors being an expression or identifier; a, b, x, y, and z for identifiers; $root, p$, and q for paths or parts there of; s, f, and $.ext$ for strings.

Properties of rules: CloudMake's semantics, as well as its data types introduce many properties that allow us to refactor and simplify expressions. For each syntactic construct and for most data types we have both introduction and elimination rules. For example, *functions* are introduced by using $\lambda-$expressions; functions are eliminated when they are applied. We call these rules Fold and Unfold, respectively, and use them to introduce and simplify functions.

$$\mathsf{Fold} :: E1([E_2 \textbf{ for } x]) \longrightarrow ((x) \texttt{ => } E_1)(E_2)$$
$$\mathsf{Unfold} :: ((x) \texttt{ => } E_1)(E_2) \longrightarrow E1[E_2 \textbf{ for } x]$$

Variables can be introduced, eliminated, and reordered. We often use variable introduction for common subexpression elimination.

$$\mathsf{IntroVar} :: S[E \textbf{ for } x] \longrightarrow \textbf{var } x = E; S[x \textbf{ for } E]$$
$$\mathsf{ElimVar} :: \textbf{var } x = E; S \longrightarrow S[E \textbf{ for } x]$$
$$\mathsf{ReorderVar} :: \textbf{var } x = E; S \longrightarrow S; \textbf{var } x = E$$

The possibility of reordering variables is essential for our matching. Each of CloudMake's statements and data types (numbers, strings, paths, arrays, and objects) defines a neutral element and an overloaded associative combine operator. For statements, the combine operator is simply concatenation; for data types we assume it to be (+), noting that in proper CloudMake we have to use different function names for the overloaded (+) operators. This allows us to express rewrite rules suc-

cinctly and to apply associative, and possibly even commutative, matching to instantiate program schemes.

Introduce relative paths: Paths describe traversals through the directory structure. CloudMake supports three literals for paths. Absolute paths start with a drive letter, as in `C:`. Relative paths start with a single / and are relative to a given root. Paths starting with a // are UNC paths. Paths without a leading slash are relative to the current directory of the CloudMake script. A root and the first location step of UNC paths are configuration parameters to a CloudMake build.

The goal of path transformations is to eliminate absolute paths and to shorten paths as much as possible. Given a source *root*, rule IntroRoot makes all possible source paths relative to the source root. Remaining paths should come from outside the root directory. Rule IntroAlias introduces aliases for all external prefixes. The rule's applicability condition assumes that the chosen alias 'q' is unique, that is, that there should be no two files $p/q/s$ and $u/q/s$ which could not be distinguished by choosing 'q' as the alias. Rule IntroArtifact introduces objects for files identified by those external aliases. As in the previous rule, the rule assumes that f does not occur in S.

IntroRoot :: $S[\text{`}root/P\text{'}\ \textbf{for}\ x] \longrightarrow S[\text{`}/P\text{'}\ \textbf{for}\ x]$
IntroAlias :: $S[\text{`}p/q/s\text{'}\ \textbf{for}\ x] \longrightarrow S[\text{`}//q/s\text{'}\ \textbf{for}\ x]$
IntroArtifact :: $S[\text{`}p/f.ext\text{'}\ \textbf{for}\ x] \longrightarrow$
$\quad \textbf{var}\ f = \{ext : \text{`}p/f.ext\text{'}\};$
$\quad S[f.ext\ \textbf{for}\ x]$

In order to support multi-directory builds, we partition the set of definitions into a set of build scripts, spread over the directory hierarchy, inverting the effect of the split by generating proper include statements. Our heuristic is to split the build scripts according to the working directory for each tool run.

`SplitModule` ::
$\quad$//original script
$\quad \textbf{var}\ x_1 = F_1(\{dir : \text{`}/p\text{'}\} + E_1); \ldots$
$\quad \textbf{var}\ x_n = F_n(\{dir : \text{`}/p\text{'}\} + E_n);$
$\quad S \longrightarrow$
$\quad$//script '$/p/MakeScript$'
$\quad \textbf{var}\ x_1 = F_1(E_1[\text{`}q_1\text{'}\ \textbf{for}\ \text{`}/p/q_1\text{'}]); \ldots$
$\quad \textbf{var}\ x_n = F_n(E_n[\text{`}q_n\text{'}\ \textbf{for}\ \text{`}/p/q_n\text{'}]);$

$\quad$//original script
$\quad ///$ < *reference path* = '$/p/MakeScript$' >
$\quad S$

After applying these rules systematically, there are no absolute paths left; the remaining parts are as short as possible. In fact, because the execution trace preserves working directories, all generated descriptions end up in the directory they initially came from.

Introduce tools: Every execution trace uses a limited set of tools. As a refactoring step we can introduce automatic abstractions for all tools. To do so we can simply partition all `exec` calls based on the value of their `tool` field. Rule IntroTool matches an `exec` call with a particular tool and replaces it with the proper function call. The rule assumes that the input tool is described by a variable as introduced earlier by rule IntroArtifact.

IntroTool ::
$\quad S[\texttt{exec}(\{tool : f.exe\} + E)\ \textbf{for}\ x] \longrightarrow$
$\quad \textbf{var}\ F = (z)\ \texttt{=>}\ \texttt{exec}(\{tool : f.exe\} + z);$
$\quad S[F(E)\ \textbf{for}\ x)]$

Having introduced the tools, Metamorphosis specializes the introduced functions. As an applicability precondition, we require that all of F's applications have the same constant value E_x as its parameter.

SpecializeCall ::
$\quad \textbf{var}\ F = (z)\ \texttt{=>}\ E_z; S[F(\{x : E_x\} + E)\ \textbf{for}\ y] \longrightarrow$
$\quad \textbf{var}\ F = (z)\ \texttt{=>}\ E_z[E_x\ \textbf{for}\ z.x]; S[F(E)\ \textbf{for}\ y]$

Note that if E_z does not refer to $z.x$, the parameter $\{x : E_x\}$ was superfluous in the first place. In this situation we can weaken the applicability condition of the above rule and apply it irrespective of x's value.

Composition: Builds often use repeated patterns; for instance after compiling a C# file, its resulting DLL is often passed to a static analyzer, in .NET called `fxCop`. Rule Composition below introduces a new function that captures the combined effect; it replaces both calls by the new call, adopting the return values in the process.

Composition ::
$\quad \textbf{var}\ x = F(E_x); \textbf{var}\ y = G(E_y);$
$\quad S \longrightarrow$
$\quad \textbf{var}\ H = (a, b)\ \texttt{=>}\ \{$
$\quad\quad \textbf{var}\ x = F(a); \textbf{var}\ y = G(b);\ \textbf{return}\ x + y;$
$\quad \}$
$\quad \textbf{var}\ xy = H(E_x, E_y);$
$\quad S[xy[i]\ \textbf{for}\ x[i], xy[j + K]\ \textbf{for}\ y[j]]$

where K is the length of x. The rule's applicability condition requires that x be used in G_y.

Next, we merge the parameters, to get to one parameter per call. To that end, we introduce a new helper function $+\!\!+$; we define $f +\!\!+ g$ to be like $f + g$, except that all field labels that are present in both, but whose values are different get indexed with their origin. For example, $\{x : E_l\} +\!\!+ \{x : E_r\} = \{x_l : E_l\} +\!\!+ \{x_r : E_r\}$. Projection functions *left* and *right* are the corresponding projections, e.g., $left(f + g) = f$, similarly for *right*.

Refactoring	Description	Transformations	Type
`ComposeCscCopy`	Compose csc and copy	Composition	1-to-1
`ExtractCscDependencies`	Extract dependencies common among csc invocations	Properties of rules	1-to-1
`ExtractClDependencies`	Extract dependencies common among cs invocations	Properties of rules	1-to-1
`ExtractFxCopDependencies`	Extract dependencies common among fxCop invocations	Properties of rules	1-to-1
`ExtractIclDependencies`	Extract dependencies common among Icl invocations	Properties of rules	1-to-1
`IntroduceTools`	Introduce tool for common exec invocations	Introduce tools	1-to-1
`IntroduceCopyLoops`	Introduce loop for copy invocations	Introduce iterations	1-to-1
`IntroduceDeploy`	Adjust paths	Introduce iterations, Copy elimination	1-to-1
`IntroduceTttLoops`	Introduce loop for ttt invocations	Introduce iterations	1-to-1
`ReuseCopy`	Reuse copy library function	Copy elimination	1-to-1
`ReuseCsc`	Reuse csc library function	Reuse library functions	1-to-1
`ReuseTtt`	Reuse ttt library function	Reuse library functions	1-to-1
`ReuseTypeXCopy`	Reuse xcopy library function	Reuse library functions	1-to-1
`UseLibraryDlls`	Transform paths to use library dlls	Properties of rules	1-to-1
`RemoveEmptyOptionalFields`	Remove optional fields	Properties of rules	1-to-1
`InlineCopies`	Inline copy invocations	Copy elimination	1-to-1
`ArrayCompression`	Extract common array elements	Properties of rules	1-to-1
`MergeFlavors`	Merge scripts obtained for various build flavors	Merging scripts	N-to-1
`SplitModule`	Distribute script to appropriate directory	Introduce relative paths	1-to-N

Figure 5: Refactorings for CloudMake

MergeArgs::
var $H = (a, b)$ => {
var $x = F(a)$; **var** $y = G(b)$; **return** $x + y$;
}
$S[H(E_x, E_y)$ **for** $z] \longrightarrow$
var $H = (ab)$ => {
var $x = F(left(ab))$; **var** $y = G(right(ab))$;
return $x + y$;
}
$S[H(f +\!\!+ g)$ **for** $z]$

Reuse library functions: Tools such as compilers or linkers are used repeatedly by different builds. Instead of rediscovering each tool, we use library auxiliary helper functions if available. Let us assume that each tool definition F is in a particular library and that it is defined by two functions F and its helper F_{args}.

//in file '$//tool/F$'
var $F = (x)$ => E_F
var $F_{args} = (x) = E_{Fargs}$

Function F calls the original tool definition, that is $F(E)$ will call `exec`($\{tool : \text{'}//tool/F.exe\text{'}\} + \ldots$), but in general F might have different arguments than `exec`. This is the case where the helper F_{args} comes in. It adapts `exec`'s arguments to match F's arguments. We can now map every library use of a tool to its special-purpose tool.

ReuseFun ::
var $x =$ `exec`($\{tool : \text{'}//tool/F.exe\text{'}\} + E$); $S \longrightarrow$
$/// <$ *reference path* $= \text{'}//tool/F\text{'} >$
var $x = F(F_{args}(E)); S$

By unfolding F_{args} we get to required calls that have tool specific parameter information. For instance, Metamorphosis reduces an `exec` call to `copy.exe` with its numerous parameters to a simple `copy` call, that takes as its input only the source path of the file to copy and its destination path.

Copy elimination: Builds often perform a great deal of copy operations, frequently more than is necessary. In most cases the trivial elimination rule ElimCopy takes care of this redundant copying.

ElimCopy ::
var $o = copy(s, d); S \longrightarrow S[s$ **for** $o]$

Unfortunately, we cannot always safely apply this simple rule without breaking a build. Certain tools, like `fxCop` for example, require files at certain location. If `fxCop` is the consumer of the copy operation, we are not allowed to apply this rule. We are also not allowed to apply this rule for copy operations whose output is not used by another tool in the program. Those copies are typically part of the deployment. The applicability condition of this rule thus depends on the property of the consumer.

We can sometimes eliminate copies whose source is provided by tools that place results at an expected location, as shown below:

RedirectOut::
var $t = F(\{out : [E_0, \ldots, E_n]\} + E)$;
var $o = copy(t[i], d)$;
$S \longrightarrow$
var $t = F(\{out :$
$[E_0, \ldots E_i - 1, d, E_i + 1, \ldots, E_n]\} + E)$;
$S[t[i]$ **for** $o]$

Recall that the field *out* denotes the sequence of paths for the tool's expected outputs. Here, we force the output of the preceding tool to go to the destination of the copy. This may in fact *improve* the overall performance.

Merging scripts: Builds are often parameterized with settings that determine target platforms (e.g., x86, IA64, etc.) and configurations (e.g., debug, release, continuous integration, etc.). We call a build with a specific setting a *build flavor*. For generality of the resulting build script, we need to "merge" the build scripts that are synthesized from execution traces of different build flavors. There can be a large number of build flavors due to combinations of setting values. However, according to our observations at Microsoft, product groups typically have fewer than ten flavors.

Without loss of generality, we consider merging two build scripts D and R for debug and release builds, respectively. The idea of build script merging consists of four steps:

1. Perform α-conversion on D and R such that they have disjoint sets of variable names.
2. Concatenate the scripts D and R, and denote the result of concatenation by $D \otimes R$.
3. For definitions in $D \otimes R$ that invoke the same tool and operate on the same set of input dependencies (including source files and intermediate outputs),
   (a) define a new binding with joint effect, and
   (b) substitute definitions with the newly introduced bindings.
4. Simplify arguments of the new bindings by pushing inwards the conditional expressions resulting from merging expressions.

As we will see, the substitution in step (3) can make some definitions unused, so they can be removed from the resulting build script.

In describing our rewrite rules below, we focus on step (3). We assume that the α-conversion has been performed such that all variables coming from D and R are subscripted with, respectively, D and R. We also assume that the concatenation has been performed.

We consider three cases in merging build scripts: isomorphic tool invocations, tool invocations with the same dependencies but different arguments, and injection of intermediate steps. We say that two tool invocations are isomorphic if they invoke the same tool with the same set of arguments, including input dependencies. The following simple rule is used to merge isomorphic tool invocations:

MergeIsomorphicTools ::
var $x_D = F(e)$;
var $x_R = F(e)$;
$S \longrightarrow$
var $x = F(e)$;
$S[x$ **for** x_D, x **for** $x_R]$

Note that the above rule removes the unused definitions x_D and x_R as well. The requirement of operating on the exact same set of input dependencies can be further relaxed by introducing a threshold for similarities of sets of input dependencies.

Different build flavors can invoke the same tool with the same set of input dependencies, but with different arguments. For example, in the debug build the `csc` compiler is called with `/debug+` option, while in the release build, to achieve high performance, the compiler is called with `/debug-`. We assume that the flag *debug* indicates that the debug build is enabled, and is in the scope. The following rewrite rule can be used to merge such tool invocations:

MergeDiffArgTools ::
var $x_D = F(\{deps : [s_1, \ldots, s_n]\} + E_D)$;
var $x_R = F(\{deps : [s_1, \ldots, s_n]\} + E_R)$;
$S \longrightarrow$
var $x = F(\{deps : [s_1, \ldots, s_n]\} + debug?E_D : E_R)$;
$S[x$ **for** x_D, x **for** $x_R]$

Note that each input dependency s_i can be either a source file, or a variable (denoting an intermediate file) resulting from previous merging.

The formalization of the third case in merging build scripts, as well as **Introduce iterations** transformation is available in the appendix (Section A).

4.2 CloudMake Refactorings

Figure 5 lists the refactorings implemented in Metamorphosis. For each refactoring, we provide a short description ("Description" column) and a list of transformations that the refactoring uses ("Transformations" column). In addition, we specify the type of each refactoring ("Type" column). The type is determined based on the number of input/output build scripts; 1-to-1, 1-to-N, and N-to-1 types specify that a refactoring takes one/one/N script(s) as input and produces one/N/one script(s) as output, respectively. (We illustrate several refactorings in the appendix in Figure 11.)

It is important to note that all currently supported refactorings are idempotent by design. More importantly, we manually identified commutativity relation among refactorings; overall 72% of all refactoring pairs commute. (Because of the space limit, we show the commutativity relation in the appendix in Figure 12.) We consider idempotence and commutativity while optimizing the search for the best build script (according to a fitness function). As shown in Section 6, these optimizations lead to significantly faster exploration.

5. Phase Two: Exploration

In this section, we describe the second phase of our migration approach. Figure 4 (right part) illustrates this

phase. The inputs to this phase are synthesized build scripts (the output of the first phase). The outputs are refactored scripts, located in appropriate directories.

To detect a refactoring sequence that gives the best resulting script, based on some criteria (e.g., minimal number of nodes in the AST, etc.), we perform search-based refactoring [55, 61]. In the rest of the text, we refer to the content of build scripts under refactoring as a *state*. Figure 6 shows the pseudo-code for an algorithm for finding the "best" state. We first describe the *naïve* algorithm (Figure 6 without highlighted lines).

Search algorithm: The inputs to the algorithm are "flat" synthesized scripts p (which is the initial state), a fitness function F, and a maximum search depth D. We use a priority queue (line 4) to keep the states that should be further explored; the priority is determined by the fitness. Each element in the queue saves a refactoring sequence, a script obtained by executing the sequence, and the length of the sequence. The algorithm proceeds as long as there are elements in the queue (line 7) and ignores sequences longer than the given maximum length (line 9). Next, the algorithm modifies the current state (that has the highest fitness value) with each refactoring available in the set of all refactorings (line 11). If a refactoring is applicable and the type of the refactoring (e.g., 1-to-1) matches the current state (line 12), the refactoring is applied, the fitness value of the new state is calculated, and the refactoring is appended to the refactoring sequence. If the fitness value of the new state is better than previous best fitness value (line 22), the algorithm saves the new value and the refactoring sequence that leads to that value.

While simple, this naïve algorithm suffers from a great challenge – state explosion. The algorithm is inherently exponential. We propose an optimized search-based refactoring algorithm that uses state matching and partial-order reduction, two optimizations commonly deployed in model-checking [29, 34, 41, 43].

State matching: Considering that several (independent) refactoring sequences may lead to the same state, we can optimize the search by performing state matching. Namely, whenever a new state is encountered, we check if the state has been previously seen. If the state has been seen, we stop the exploration of the current refactoring because the resulting script from a state s is the same regardless of the sequence that led to s. To perform state matching, we need to save all the states that have been encountered. As a way to reduce the memory footprint, we replace the explored state of an exploration tree discovered by Search with its hash value, relinquishing the memory that is required to hold the entire state.

Partial-order reduction: In addition to state matching, we also reason about commutativity of refac-

```
1  P = ∅                                   ▷ Visited scripts
2  R = ∅                                   ▷ Visited sequences
3  function Search(p, F, D)                ▷ Project p, fitness F, depth D
4      Q⟨f, r⃗, p, d⟩ ← [⟨0, [], p, 0⟩]     ▷ Priority queue based on f
5      f_max = 0
6      r⃗_best = []                         ▷ Best discovered sequence
7      while Q ≠ ∅ do
8          t = Q.pull                      ▷ Take tuple with highest priority
9          if t.d > D then continue        ▷ Depth check
10         end if
11         for all ⟨r_i⟩ ∈ Refactorings do
12             if IsApplicable(r_i, t.p) ∧ τ(r_i) = τ(t.p) then
13                 p′ = r_i(t.p)           ▷ Apply refactoring
14                 r⃗′ = t.r⃗ • r_i          ▷ Add last refactoring
15                 r⃗_0 = GetCanonical(r⃗′)   ▷ Get canonical form
16                 if p′ ∈ P ∨ r⃗_0 ∈ R      ▷ Check if explored
17                     continue
18                 end if
19                 P = P ∪ {p}             ▷ Update explored sets
20                 R = R ∪ {r⃗}
21                 f = F(p′)               ▷ Get the fitness
22                 if f > f_max then       ▷ Check for better fitness
23                     f_max = f           ▷ Save best fitness
24                     r⃗_best = r⃗′          ▷ Save best sequence
25                 end if
26                 Q.insert(⟨f, r⃗′, p′, d + 1⟩)
27             end if
28         end for
29     end while
30     return r⃗_best
31 end function
```

Figure 6: Search-based refactoring approach; optimized version includes highlighted lines

torings [34]. As mentioned earlier, all refactorings in Metamorphosis are idempotent. Furthermore, 72% of all refactoring pairs commute. These algebraic properties allow us to significantly reduce the search space, as we only need to explore some refactoring sequences and can prune others without the danger of missing the highest-fitness solution.

The pseudo-code in Figure 6 (including highlighted lines) captures the optimization ideas we have implemented in Metamorphosis. In this improved algorithm, we keep track of both the visited states in P *and* the refactoring sequences $r_1, \ldots, r_n$ in R that we have explored thus far.

Canonization of refactoring sequences on line 15 takes care of sequences that commute. For example, refactoring sequence r_1, r_7, r_3, r_3 will be reduced by the canonization operation to r_1, r_3, r_7 if r_3 is idempotent and if r_3 and r_7 commute. This allows us to significantly reduce the search space to be explored.

Parallelizing search: Metamorphosis implements a parallel version of the exploration algorithm, shown in Figure 6, using the Task Parallel Library [13] (TPL). Each refactoring invocation is executed as a single task in a task pool. The drawback of using the default configuration of TPL is the lack of control over the task scheduler. To control the priorities of tasks and to enforce parallel search, we implemented our own task

Project	Build System	Size (KB)	AST Nodes	Deps.	Tools	Tools Invoc.	csc Invoc.	copy Invoc.	Env. Vars	Input Scripts
Internal tool	CloudMake	431	13,668	2,279	5	94	69	0	0	71
Synthetic	MSBuild	3,221	143,331	22,770	2	400	200	200	0	200
Cloud managment	NMake	8,935	187,959	51,848	39	1,760	0	0	916	1,034
Analytics engine	MSBuild	15,962	419,476	52,462	28	16,643	433	14,590	562	575

Figure 7: Characteristics of projects and synthesized build scripts

scheduler that greedily gives priority to the tasks that work on states with higher fitness values.

Our parallel runner is configurable in that one can specify the number of threads in the pool, maximum concurrency allowed by the task scheduler, and processor affinity. While limiting the number of processors was mostly done for experimental proposes, limiting the number of threads in the pool was in fact necessary to limit the memory consumption when the script on which exploration is invoked is large.

Note that we constrain the search process to a small degree by enforcing the structure (Figure 4). First, different build flavors are merged into one. Then additional (1-to-1) refactorings take place. Finally, the script is split across the directory hierarchy. The best refactoring sequence in the second step is discovered through the search algorithm outlined in Figure 6.

Fitness functions: In our current implementation of Metamorphosis, we have experimented with several fitness functions, but ultimately used the aggregate length of build script (i.e., number of characters) as our metric. (We also used the number of AST nodes, but we observed no difference in our results.) In the future, we plan to experiment with other fitness functions that combine both the overall size of the scripts and a measure of their complexity and maintainability.

6. Evaluation

The goal of our experimental evaluation was to answer the following research questions:

*RQ*1. Is Metamorphosis able to synthesize build scripts for existing builds, regardless of the build system used and tools invoked, while containing thousands of tool invocations?

*RQ*2. What are the benefits of our optimized search-based refactoring algorithm?

*RQ*3. Does the search-based refactoring algorithm, with 17 refactorings, lead to maintainable CloudMake build scripts?

We performed all the experiments on a machine with 32 cores, Intel Xeon CPU E5-2650 @ 2.00Hz, 64 GB of RAM, 2 SSDs (in RAID), running Windows Server 2012.

6.1 Projects

Figure 7 lists the projects used in our evaluation[3].

- ⋆ Internal tool is our build system that parses CloudMake files and runs them in the cloud. Build scripts for Internal tool are written in CloudMake.
- ⋆ Synthetic is a relatively small, synthetically-generated project, which has build scripts written in MSBuild. This project was generated as one of the test cases for CloudMake (independently of our study).
- ⋆ Cloud managment is a large project that uses NMake and relies on 39 different tools.
- ⋆ Analytics engine is a very large project that uses MSBuild and relies on 28 different tools.

With this selection of projects, we aimed to achieve diversity in terms of build systems used (second column in Figure 7) and also in terms of size, ranging from modestly-size projects all the way to large projects with millions lines of code.

6.2 *RQ*1 Synthesis

Metamorphosis successfully synthesized build scripts for all projects used in the evaluation.

Figure 7 shows the size of synthesized build script in column 3; the size can be as large as 15 MB. Note that in cases where multiple build flavors were present, we measured properties of one of them, as scripts for various build flavors are generally similar to each other. The number of AST nodes is displayed in column 4; the number of AST nodes indicates a scale of scripts transformed by refactorings. The number of dependencies (i.e., input files that are outputs of other tools) between the tools is shown in column 5. The number of unique tools used in the build is shown in column 6 (these typically correspond to build tools like the C# compiler `csc` or the C++ compiler `cl`). Column 7 shows the total number of tool invocations. The number of invocations is perhaps most instructive in understanding the scale of these projects, as it captures the number of `exec` invocations that need to be run for the build to finish. In columns 8 and 9 we also show the num-

[3] Due to policy concerns, we anonymize project names.

Project (Search technique)	Exploration Time	Fitness (number of chars) Original	Best	Reduction	Unique States	Applied Refactorings
Internal tool(Stateless)	3 sec	402,610	249,206	38%	17	23
Internal tool(Stateful)	3 sec	402,610	249,206	38%	17	22
Internal tool(Stateful+POR)	3 sec	402,610	249,206	38%	17	22
Synthetic(Stateless)	> 1 h	3,020,687	1,619,200	46%	245	3,845
Synthetic(Stateful)	18 min	3,020,687	1,619,200	46%	654	1,321
Synthetic(Stateful+POR)	15 min	3,020,687	1,619,200	46%	618	1,086
Cloud managment(Stateless)	43 sec	9,138,210	7,284,348	20%	15	21
Cloud managment(Stateful)	37 sec	9,138,210	7,284,348	20%	15	21
Cloud managment(Stateful+POR)	38 sec	9,138,210	7,284,348	20%	15	21
Analytics engine(Stateless)	> 2 h	15,764,073	9,155,304	41%	273	3,307
Analytics engine(Stateful)	> 2 h	15,764,073	8,744,943	44%	1,388	3,320
Analytics engine(Stateful+POR)	> 2 h	15,764,073	8,699,800	45%	1,741	3,422

Figure 8: Exploration times and other statistics

ber of calls to the C# compiler `csc` and the number of `copy` invocations; several of our refactorings are related to these tools. Several tools have dependencies on environment variables, as captured in column 10. Finally, the scale of the used projects is also reflected in the number of directories that contain sources, as shown in column 11; each directory contains one build script.

6.3 *RQ2* Exploration

Our proposed refactorings reduce build script size by up to 46%. Further, optimized search-based refactoring algorithm may explore up to 6.3× more *unique states* than the naïve algorithm, in the same amount of time.

Figure 8 shows some of the basic statistics for the automatic exploration of refactoring sequences. (We used a single core for these runs.) In column 1 we show names of search techniques: *Stateless* which is naïve exploration without any optimization (algorithm in Figure 6 without highlighted lines), *Stateful* which performs state matching (as explained in Section 5), and *Stateful+POR* which corresponds to the highest degree of optimization (algorithm in Figure 6 with highlighted lines). Column 2 shows the exploration time for each project and search technique; exploration times vary significantly, depending on how applicable the refactoring techniques turn out to be, ranging from a few seconds to over 2 hours. It should be noted that Most of the refactorings have been inspired by Analytics engine. As we work on migrating build scripts of other projects, we expect to introduce more refactorings that are widely applicable. Columns 3–5 show the starting and final fitness values, which in our case are the number of characters in the CloudMake scripts, as well as the reduction in the number of characters as a percentage, ranging between 20–46%. Note that we consider the reduction for a sequence of 1-to-1 refactorings (Section 5). Column 6 shows the number of unique states encountered during the exploration. Column 7 shows the total

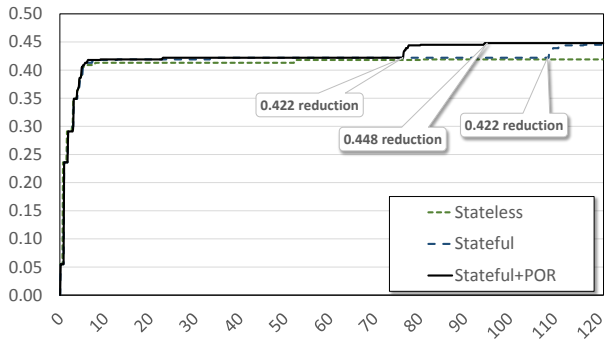

Figure 9: Reduction with optimized and naïve exploration (y-axis) compared over two hours (x-axis)

number of executed refactorings. These two quantities correlate well with the overall exploration time.

It is important to observe in Figure 8 that the most optimized algorithm, i.e., Stateful+POR, explores 6.3× more unique states than the naïve algorithm. Figure 9 shows reduction in build script size achieved by different algorithms (*Stateless*, *Stateful*, and *Stateful+POR*) over a period of 2 hours for Analytics engine project. Overall, we see that the more optimized algorithms get to higher fitness levels (shown as reductions in size) faster, i.e., in 94 minutes instead of 120 minutes. Note that a high fitness value is achieved relatively fast in all cases. This happens because most of currently supported refactorings commute and even naïve search can obtain one sequence that provides good result. However, as we introduce new refactorings, finding a refactoring sequence that leads to the script with the "best" fitness value will become harder.

To confirm that finding the best refactoring sequence is challenging if numerous refactorings are available, we asked 8 professional developers (familiar with CloudMake) at Microsoft to use our refactorings and provide

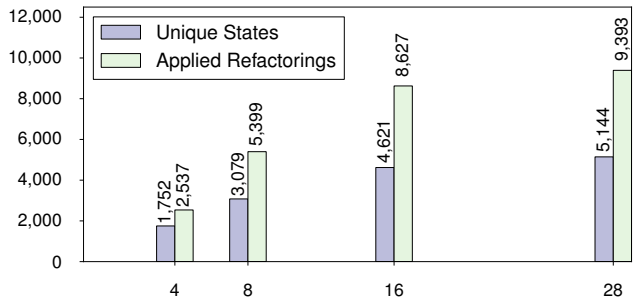

Figure 10: The number of unique states explored and refactorings applied (y-axis) in two hours as more CPUs are used (x-axis)

the refactoring sequence that would lead to the "best" (in terms of the number of characters) build script for Analytics engine. We limited the time for each developer to 15 minutes. Interestingly, no developer proposed the sequence that exploration obtained. The best and the worst reduction by developers' refactoring sequences were 37% and 16%, respectively. As we introduce new refactorings, the advantage of optimized exploration becomes evident.

We also explore the value of parallelism in optimizing search-based refactoring (discussed in Section 5). To demonstrate the value, we run our optimized exploration of Analytics engine with a different number of CPU cores used. Figure 10 shows how the exploration scales with the number of cores. The run with 28 cores consumed most of the available memory – as the synthesized script being refactored is large and the exploration maintains a number of scripts at the same time – which leads to limited exploration speed. (Running the exploration with 32 cores runs out of memory.) Comparing the number of explored states (1,741) on a single core (Figure 8) vs. multiple cores, we observe that the exploration was not (or cannot be) parallelized ideally (due to scheduling, synchronization, etc.). However, we still obtained significant speedups, up to 3×.

6.4 *RQ*3 Maintainability

We performed a *simple preliminary study* to measure how actual developers perceive the results of the migration and refactoring. Specifically, we interviewed a developer who ported Internal tool from MSBuild to CloudMake, and we also performed a survey with several developers familiar with CloudMake.

The interviewee compared several refactored scripts and his handcrafted CloudMake scripts. Although the scripts were almost exactly the same (e.g., the scripts were hierarchically organized in the same manner, all relative paths were the same, and build flavors were merged the same way), there were minor differences due to several missing refactorings. The most notable missing refactorings are those that introduce uses of existing library functions (as well as their composition) and introduction of variables that keep shared expressions.

We also performed a survey, where we asked eight professional developers to rank the maintainability of refactored script snippets (similar to Figure 11) on a scale from 1 to 10 (over 5 means "easier to maintain than the original script"). We provided one snippet for each refactoring. The snippets used in the study were produced by Metamorphosis on one of the projects used in our experiments. Only one refactoring (`ComposeCscCopy`) was ranked, on average, below 5 (3.7), while the average for other 16 refactorings was higher (6.3). In the future, we plan to perform controlled experiments to evaluate maintainability.

7. Threats to Validity

External: The main threat to external validity is that our results may not be generalizable to other build systems. Although our refactorings are language-specific, we believe that other parts of our migration approach are general. Specifically, capturing execution traces is independent of the original build system. Also, finding the best refactoring sequence can be applied to any set of refactorings (e.g., Java refactorings for code not build scripts) where one can define a fitness function (e.g., the length of the refactored code).

Internal: The main threat to internal validity is the potentially incorrect implementation of Metamorphosis. To mitigate this threat, we tested our implementation thoroughly. Several tests execute the original and synthesized build scripts and verify that the resulting binaries are the same. Furthermore, Metamorphosis went through a rigorous review process that follows accepted engineering and coding standards at Microsoft.

Construct: Several of the refactoring preconditions are specific to the script format produced by the migration. Although these preconditions do not influence our results, they may be a limitation when the refactorings become part of CloudMake. Limit for the exploration time was set to two hours. Waiting longer than two hours did not yield different conclusions in our experiments. Finally, our migration approach is entirely based on dynamic analysis of the original build runs. One can argue that static analysis may lead to better results (with less effort). However, static approaches have several disadvantages. We list two of them here. First, so called in-proc tasks (i.e., the tasks that are executed by the build system itself without running an external process) cannot be detected and detecting all implicit dependencies between tools is difficult. Second, static approaches are tightly couped to the original build system. Last but not least, our dynamic approach may remove dead code and unused options.

8. Related Work

We provide an overview of migration techniques, refactoring literature, and search-based software engineering.

Build migration: Migration from one build system to another is most commonly done with the goal of improving performance and maintainability [17]. Complexity of build scripts is high [16, 17, 47, 48]; therefore, manual migration may lead to a number of challenges and often to failures. Suvorov et al. [62] and Neundorf [52] studied migration of two large open-source projects: Linux and KDE. The studies report that a year was spent on a failed (manual) migration for Linux kernel v.2.5 and another year was spent on successful (manual) migration for v.2.6. The KDE project offers similar experiences. Few approaches explored automatic migration using static analysis of build scripts [5]. However, these approaches do not discover all dependencies, which have to be explicitly specified for cloud-based build systems. We propose the first migration approach based on dynamic analysis that captures all the dependencies.

Fabricate [3] and Memoize [9] monitor all dependencies (i.e., opened files) when a given command is executed. Note however that these tools give only a single list of dependencies, not a computation graph; without the entire graph we cannot enable parallelism.

Build maintenance: Refactorings have been first studied over a decade ago [56, 59], and attracted attention from both practitioners and researchers [28, 33, 58, 60]. Integrated development environments (such as VisualStudio, Eclipse, NetBeans, and IntelliJ) support popular refactorings [44, 50, 65]. Recently, several refactorings were proposed to retrofit existing sequential code to use concurrent constructs [28, 60]. These refactorings improve performance and target general-purpose programming languages. We propose the most extensive list of refactorings for build scripts so far; few proposed refactorings (e.g., `InlineCopies`) can improve performance of the builds.

A few projects have explored maintenance of build scripts [16, 37, 64, 66]. Formiga [37] supports simple renaming and removal of targets. MAKAO [16] uses an aspect-oriented approach to support adding new commands, dependencies, etc. Although MAKAO extracts dependencies from execution trace, it has false positives and false negatives. SYMake [64] focuses on renaming and extraction of targets. Vakilian et. al [66] developed tools for decomposing Google build specifications. Although valuable, previously proposed refactorings are not suitable for improving synthesized build scripts. We propose the most extensive list of refactorings inspired by common patterns, which significantly reduce the size of automatically synthesized build scripts. Also, we applied our refactorings on large industrial projects.

Search-based software engineering: Discovering a "good" sequence of transformations has been explored in other domains: 1) improving software design [15, 18, 21–23, 31, 32, 38, 39, 49, 54, 55, 57, 61]; 2) improving performance [26, 53, 67, 68]; 3) fixing bugs [35, 46] and tests [27]; and 4) deriving formally correct functional programs from operational specifications [36].

Whitfield and Soffa [68] created a framework for exploring compiler optimizations. Abdeen et al. [15] proposed a search technique based on simulated annealing to optimize the package structure of source code. Hill climbing has been used for cost estimation [45] and applications of modularization [49]. Genetic algorithms (GA) and clustering methods have been used to reduce sizes of libraries [18, 30]. Fatiregun et al. [31] defined transformation problems as a search for optimization and showed that GAs outperform hill climbing. Cooper et al. [26] apply biased random search to detect a sequence that leads to a minimal value of an fitness function. Nisbet [53] applied GA to search for a sequence of transformations that would have optimal execution on parallel architectures. Forrest et al. [35] proposed program repair using GAs. Harman [38] introduced a technique, named "testability refactoring", which searches for sequences of refactorings that makes code testable and maintainable at the same time. White et al. [67] used GA to improve non-functional properties of programs, such as execution time.

To the best of our knowledge, we are the first to optimize the exploration by using partial-order reduction in search-based refactoring algorithms. In addition, we parallelize the search and evaluate its scalability.

9. Conclusions and Future Work

We have developed an automatic approach that uses dynamic analysis for migrating build scripts from any build system to CloudMake. Our approach works in two phases. First, we execute the original build, monitor the execution, and synthesize a new build script from execution traces. Second, we use search-based refactoring to discover a refactoring sequence that leads to the best build script (according to a fitness function). As search in the space of possible refactoring sequences is a formidable task, we optimize search through the use of parallelism and partial-order reduction. We implemented our approach in a tool called Metamorphosis and configured the tool to use 17 refactorings for CloudMake that we developed.

We have applied Metamorphosis to large software projects at Microsoft, some of which contain over 1,000 build scripts written for various build systems. The results show that Metamorphosis can reduce size of synthesized scripts up to 46%. Further, our optimized

search may explore up to 6.3× more refactoring sequences than naïve search in the same amount of time.

In future, we plan to define additional refactorings and explore how they generalize to other build systems. Also, we believe a hybrid approach that does some static analysis of the original build scripts in addition to dynamic analysis will enhance the end-result.

Acknowledgments

We thank Adrian Bonar, Nick Carlson, Jacek Czerwonka, John Erickson, Dmitry Goncharenko, Baris Kasikci, Micah Koffron, Darko Marinov, Davide Massarenti, Val Menn, Kivanc Muslu, and Seva Titov as well as several other Microsoft developers for productive discussions about this work. We also thank Lamyaa Eloussi, Alex Gyori, Farah Hariri, Yun Young Lee, Owolabi Legunsen, Jessy Li, Yu Lin, Qingzhou Luo, Aleksandar Milicevic, August Shi, and Mohsen Vakilian for their feedback on a draft of this paper. The first author was an intern at Microsoft and Microsoft Research during Summer 2013.

References

[1] Ant home page. `http://ant.apache.org/`.

[2] Buck home page. `http://facebook.github.io/buck/`.

[3] Fabricate home page. `https://code.google.com/p/fabricate/`.

[4] Build in the cloud. `http://google-engtools.blogspot.com/2011/08/build-in-cloud-how-build-system-works.html`.

[5] Gradle build init plugin. `http://www.gradle.org/docs/current/userguide/build_init_plugin.html`.

[6] MSBuild home page. `http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx`.

[7] Make home page. `http://www.gnu.org/software/make/manual/html_node/index.html`.

[8] Maven home page. `http://maven.apache.org/`.

[9] Memoize home page. `https://github.com/kgaughan/memoize.py`.

[10] NMake home page. `http://msdn.microsoft.com/en-us/library/ms930369.aspx`.

[11] Ninja home page. `https://martine.github.io/ninja/`.

[12] SCons home page. `http://www.scons.org/`.

[13] TPL home page. `http://msdn.microsoft.com/en-us/library/dd537609.aspx`.

[14] Vesta home page. `http://www.vestasys.org/`.

[15] H. Abdeen, S. Ducasse, H. Sahraoui, and I. Alloui. Automatic package coupling and cycle minimization. In *WCRE*, pages 103–112, 2009.

[16] B. Adams, H. Tromp, K. D. Schutter, and W. D. Meuter. Design recovery and maintenance of build systems. In *ICSM*, pages 114–123, 2007.

[17] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. The evolution of the Linux build system. *Electronic Communications of the ECEASST*, 8:1–16, 2008.

[18] G. Antoniol, M. Di Penta, and M. Neteler. Moving to smaller libraries via clustering and genetic algorithms. In *CSMR*, page 307, 2003.

[19] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *METRICS*, pages 292–303, 1999.

[20] F. L. Bauer, R. Berghammer, M. Broy, W. Dosch, F. Geiselbrechtinger, R. Gnatz, E. Hangel, W. Hesse, B. Krieg-Brückner, A. Laut, T. Matzner, B. Möller, F. Nickl, H. Partsch, R. Pepper, K. Samelson, M. Wirsing, and H. Wössner, editors. *The Munich Project CIP: The Wide Spectrum Language CIP-L*, volume 183. 1985.

[21] T. Bodhuin, G. Canfora, and L. Troiano. SORMASA: A tool for suggesting model refactoring actions by metrics-led genetic algorithm. In *WRT*, pages 23–24, 2007.

[22] S. Bouktif, G. Antoniol, E. Merlo, and M. Neteler. A novel approach to optimize clone refactoring activity. In *GECCO*, pages 1885–1892, 2006.

[23] M. Bowman, L. Briand, and Y. Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *Transactions on Software Engineering*, 36(6): 817–837, 2010.

[24] M. Christakis, K. R. M. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *FM*, pages 643–657, 2014.

[25] D. Coetzee, A. Bhaskar, and G. Necula. A model and framework for reliable build systems. Technical Report UCB/EECS-2012-27, University of California at Berkeley, 2012.

[26] K. D. Cooper, D. Subramanian, and L. Torczon. Adaptive optimizing compilers for the 21st century. *J. Supercomput.*, 23(1):7–22, 2002.

[27] B. Daniel, T. Gvero, and D. Marinov. On test repair using symbolic execution. In *ISSTA*, pages 207–218, 2010.

[28] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential Java code for concurrency via concurrent libraries. In *ICSE*, pages 397–407, 2009.

[29] A. F. Donaldson and A. Miller. Extending symmetry reduction techniques to a realistic model of computation. *Electronic Notes in Theoretical Computer Science*, 185: 63–76, 2007.

[30] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *STEP*, page 73, 1999.

[31] D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In *SCAM*, pages 66–75, 2004.

[32] D. Fatiregun, M. Harman, and R. M. Hierons. Search-based amorphous slicing. In *WCRE*, pages 3–12, 2005.

[33] A. Feldthaus and A. Møller. Semi-automatic rename refactoring for JavaScript. In *OOPSLA*, pages 323–338, 2013.

[34] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[35] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues. A genetic programming approach to automated software repair. In *GECCO*, pages 947–954, 2009.

[36] W. Guttmann, H. Partsch, W. Schulte, and T. Vullinghs. Tool support for the interactive derivation of formally correct functional programs. *Journal of Universal Computer Science*, 9(2):173–188, 2003.

[37] R. Hardt and E. Munson. Ant build maintenance with Formiga. In *RELENG*, pages 13–16, 2013.

[38] M. Harman. Refactoring as testability transformation. In *ICSTW*, pages 414–421, 2011.

[39] M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *GECCO*, pages 1106–1113, 2007.

[40] Y. Higo, S. Kusumoto, and K. Inoue. A metric-based approach to identifying refactoring opportunities for merging code clones in a Java software system. *J. Softw. Maint. Evol.*, 20(6):435–461, 2008.

[41] G. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[42] G. Hunt and D. Brubacher. Detours: Binary interception of Win32 functions. In *USENIX Windows NT Symposium*, pages 135–143, 1998.

[43] R. Iosif. Exploiting heap symmetries in explicit-state model checking of software. In *ASE*, pages 254–261, 2001.

[44] M. Kim, T. Zimmermann, and N. Nagappan. A field study of refactoring challenges and benefits. In *FSE*, pages 1–11, 2012.

[45] C. Kirsopp, M. J. Shepperd, and J. Hart. Search heuristics, case-based reasoning and software project effort prediction. In *GECCO*, pages 1367–1374, 2002.

[46] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: fixing 55 out of 105 bugs for $8 each. In *ICSE*, pages 3–13, 2012.

[47] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of ANT build systems. In *MSR*, pages 42–51, 2010.

[48] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An empirical study of build maintenance effort. In *ICSE*, pages 141–150, 2011.

[49] B. S. Mitchell and S. Mancoridis. Using heuristic search techniques to extract design abstractions from source code. In *GECCO*, pages 1375–1382, 2002.

[50] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297, 2009.

[51] A. Neagu. What is wrong with Make?, Jul 2005. `http://freecode.com/articles/what-is-wrong-with-make`.

[52] A. Neundorf. Why the KDE project switched to CMake - and how., Apr 2012. `http://lwn.net/Articles/188693/`.

[53] A. Nisbet. GAPS: A compiler framework for genetic algorithm (GA) optimised parallelisation. In *HPCN*, pages 987–989, 1998.

[54] M. O'Keeffe and M. O'Cinneide. Search-based software maintenance. In *CSMR*, pages 249–260, 2006.

[55] M. O'Keeffe and M. O'Cinneide. Search-based refactoring: an empirical study. *J. Softw. Maint. Evol.*, 20(5):345–364, 2008.

[56] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[57] O. Räihä. A survey on search-based software design. *Computer Science Review*, 4(4):203–249, 2010.

[58] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev. Refactoring with synthesis. In *OOPSLA*, pages 339–354, 2013.

[59] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[60] M. Schäfer, M. Sridharan, J. Dolby, and F. Tip. Refactoring Java programs for flexible locking. In *ICSE*, pages 71–80, 2011.

[61] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO*, pages 1909–1916, 2006.

[62] R. Suvorov, M. Nagappan, A. Hassan, Y. Zou, and B. Adams. An empirical study of build system migrations in practice: Case studies on KDE and the Linux kernel. In *ICSM*, pages 160–169, 2012.

[63] R. Tairas and J. Gray. Increasing clone maintenance support by unifying clone detection and refactoring activities. *Inf. Softw. Technol.*, 54(12):1297–1307, 2012.

[64] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. Nguyen. SYMake: a build code analysis and refactoring tool for makefiles. In *ASE*, pages 366–369, 2012.

[65] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *ICSE*, pages 233–243, 2012.

[66] M. Vakilian, R. Sauciuc, Morgenthaler, J. D., and V. Mirrokni. Automated decomposition of build targets. Technical report, University of Illinois at Urbana-Champaign, 2014.

[67] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *Trans. Evol. Comp*, 15(4):515–538, 2011.

[68] D. Whitfield and M. L. Soffa. An approach for exploring code-improving transformations. *Trans. Program. Lang. Syst.*, 19(6):1053–1084, 1997.

A. Formalization Continued

In this section, we expand the set of transformations introduced in Section 4.1.

Introduce iterations: Arrays are introduced by array enumeration and concatenation (+), elimination is indexing. METAMORPHOSIS uses maps over arrays to represent iterative behavior. We introduce loops at the expression and the statement level:

IntroLoopExp ::
$[F(E1), \ldots, F(E_n)] \longrightarrow$
$[E1, \ldots, E_n].map(x \texttt{ => } F(x))$
IntroLoopStm ::
var $x_1 = F(E1); \ldots;$ **var** $x_n = F(E_n);$
$S \longrightarrow$
var $xs = [E1, \ldots, E_n].map(x \texttt{ => } F(x));$
$S[xs[i]$ **for** $x_i]$

After systematically applying these two rules the refactored program has no further repetition.

Merging scripts (continued): Some build flavors can inject intermediate steps. For example, a continuous-integration build may enables static analysis like code contracts. However, one can easily track the input dependencies of those intermediate steps. The following rules can be used to merge this particular case:

MergeDiffArgTools ::
var $z_R = G(y);$
var $x_D = F(\{deps : [s_1, \ldots, s_n, y]\} + E_D);$
var $x_R = F(\{deps : [s_1, \ldots, s_n, z_R]\} + E_R);$
$S \longrightarrow$
var $x = F(\{deps : [s_1, \ldots, s_n, debug?y : z_R]\}$
$+debug?E_D : E_R);$
$S[x$ **for** x_D, x **for** $x_R]$

Note that, like s_i, the input dependency y can be either a source file, or a variable (denoting an intermediate file) resulting from previous merging. In the above rule, the release build script R contains additional intermediate step, i.e., calling tool G on argument y.

We can achieve even better result by pushing the case distinction inside expressions. The following rules relating conditionals and objects allow us to push the case distinction inside the object or eliminate it completely. (The rules for arrays, strings, and paths are similar.)

ElimCond ::
$debug?E : E$
$\longrightarrow E$
DistrSameFieldSameValue ::
$debug?\{x : E_x\} + E_l : \{x : E_x\} + E_r$
$\longrightarrow \{x : E_x\} + (debug?E_l : E_r)$
DistrLeftObjectEmpty ::
$debug?\{\} : \{x : E_x\} + E_r$
$\longrightarrow \{debug?undefined : E_x\} + E_r$
DistrRightObjectEmpty ::
$debug?\{x : E_x\} + E_r : \{\}$
$\longrightarrow \{debug?E_x : undefined\} + E_r$
DistrSameFieldDifferentValue ::
$debug?\{x : E_{xl}\} + E_l : \{x : E_{xr}\} + E_r$
$\longrightarrow \{x : debug?E_{xl} : E_{xr}\} + (debug?E_l : E_r)$

Distribution rules are confluent, that is once they terminate, the resulting program distinguishes values only where needed. Merging more than two build scripts, works iteratively by merging two scripts at a time.

B. Examples of Refactorings

Figure 11 shows several examples of refactorings currently supported in METAMORPHOSIS.

C. Commutativity Relation

Figure 12 shows commutativity relation among refactorings.

Refactoring	Before	After

ComposeCscCopy	`var csc = csc({ out: projA + "obj\debug\A.dll", ...})`
	`var copy = copy(csc.dll, projA + "bin\debug\A.dll");`
		`var cscAndCopy = cscAndCopy({`
		`  out: projA + "obj\debug\A.dll",`
		`  dllDst: projA + "bin\debug\", ...`
		`}`

IntroduceTools	`var copy = exec({`
	`  dir: projA,`
	`  tool: "c:\windows...\cp.exe",`
	`  tool_args: [`
	`  csc.dll,`
	`  projA + "bin\debug\A.dll"]`
	`  deps: [csc.dll],`
	`  env: [{"PATH", programFiles + "..."}],`
	`  out: ["c:\projects...\A\bin\debug\A.dll"]`
	`})`
		`fun copy(src, dst):`
		`  exec({`
		`  dir: ".",`
		`  tool: "c:\windows...\cp.exe",`
		`  tool_args: [src, dst],`
		`  deps: [src],`
		`  env: [],`
		`  out: [dst]`
		`  })`
		`var copy = // unchanged`

IntroduceCopyLoops	`var copyDll = copy(csc.dll, projA + "bin\debug\A.dll");`
	`var copyPdb = copy(csc.pdb, projA + "bin\debug\A.pdb");`
		`var copies = [{csc.dll, projA + "bin\debug\A.dll"}`
		` {csc.pdb, projA + "bin\debug\A.pdb"}`
		`].map(s, d => copy(s, d))`

IntroduceDeploy	`var copyA = copy(csc.dll, projA + "bin\debug\A.dll");`
	`...`
	`var copyB = copy(csc.pdb, projB + "bin\debug\B.dll");`
		`...`
		`var deploy = [{csc.dll, projA + "bin\debug\A.dll"}`
		` {csc.dll, projB + "bin\debug\B.dll"}`
		`].map(s, d => copy(s, d))`

ReuseCopy	`var copy = exec({`
	`  dir: projA,`
	`  tool: "c:\windows...\cp.exe",`
	`  tool_args: [csc.dll, projA + "bin\debug\A.dll"]`
	`  deps: [csc.dll],`
	`  env: [{"PATH", programFiles + "..."}],`
	`  out: ["c:\projects...\A\bin\debug\A.dll"]`
	`})`
		`var copy = copy(csc.dll, projA + "bin\debug\A.dll");`

UseLibraryDlls	`["/reference:", "c:\program...\v4.5\mscorlib.dll"],` `["/reference:", "c:\program...\v4.5\system.core.dll"]`	`["/reference:", MsCorLib.dll],` `["/reference:", System.Core.dll]`

InlineCopies	`var csc1 = csc({ out: projA + "obj\debug\A.dll", ...})`
	`var copy = copy(csc1.dll, projA + "bin\debug\A.dll");`
	`var csc2 = csc({ references: copy, ...})`
		`var csc1 = csc({ out: projA + "obj\debug\A.dll", ...})`
		`var csc2 = csc({ references: csc1.dll, ...})`

ArrayCompression	`["/reference:", "c:\program...\v4.5\mscorlib.dll"],` `["/reference:", "c:\program...\v4.5\system.core.dll"]`	`uncompress("/reference:", MsCorLib.dll, System.Core.dll)`

MergeFlavors	`var csc1 = csc({ // Debug trace`
	`  debug: "+",`
	`  filealign: 512,`
	`  optimize: "-", ...`
	`})`
	
	`var csc2 = csc({ // Release trace`
	`  debug: "pdbonly",`
	`  filealign: 512,`
	`  optimize: "+", ...`
	`})`
		`var csc = csc({`
		`  debug: DEBUG ? "+" : "pdbonly",`
		`  filealign: 512,`
		`  optimize: DEBUG ? "-" : "+", ...`
		`})`

SplitModule	`script {`
	`  var csc1 = csc({ sources: [projA + "C.cs"] ...})`
	`  var csc2 = csc({ sources: [projC + "C.cs"] ...})`
	`}`
		`scriptA { // in dir projA`
		`  var csc1 = csc({ sources: ["C.cs"] ...})}`
		`scriptC { // in dir projC`
		`  var csc2 = csc({ sources: ["C.cs"] ...})}`

Figure 11: Examples of refactorings in METAMORPHOSIS

Refactoring	ComposeCscCopy	ExtractCscDependencies	ExtractClDependencies	ExtractFxCopDependencies	ExtractIclDependencies	IntroduceTools	IntroduceCopyLoops	IntroduceDeploy	IntroduceTttLoops	ReuseCopy	ReuseCsc	ReuseTtt	ReuseTypeXCopy	UseLibraryDlls	RemoveEmptyOptionalFields	InlineCopies	ArrayCompression
ComposeCscCopy	-	✗	✓	✓	✓	✓	✗	✗	✓	✗	✗	✓	✓	✓	✗	✗	✗
ExtractCscDependencies	✗	-	✓	✓	✓	✗	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓
ExtractClDependencies	✓	✓	-	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ExtractFxCopDependencies	✓	✓	✓	-	✓	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
ExtractIclDependencies	✓	✓	✓	✓	-	✗	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
IntroduceTools	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✗
IntroduceCopyLoops	✗	✓	✓	✓	✓	✗	-	✗	✓	✗	✓	✓	✓	✓	✓	✗	✓
IntroduceDeploy	✗	✓	✓	✓	✓	✗	✗	-	✓	✗	✓	✓	✓	✓	✓	✗	✓
IntroduceTttLoops	✓	✓	✓	✓	✓	✗	✓	✓	-	✓	✓	✗	✓	✓	✓	✓	✓
ReuseCopy	✗	✓	✓	✓	✓	✗	✗	✗	✓	-	✓	✓	✓	✓	✗	✗	✗
ReuseCsc	✗	✗	✓	✓	✓	✗	✓	✓	✓	✓	-	✓	✓	✓	✗	✓	✗
ReuseTtt	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	-	✓	✓	✗	✓	✗
ReuseTypeXCopy	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	-	✓	✗	✓	✗
UseLibraryDlls	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	-	✓	✓	✓
RemoveEmptyOptionalFields	✗	✓	✓	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✓	-	✓	✓
InlineCopies	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✓	-	✓
ArrayCompression	✗	✓	✓	✓	✓	✗	✓	✓	✓	✗	✗	✗	✗	✓	✓	✓	-

Figure 12: Commutativity relation among refactorings in Metamorphosis