Lightweight, semi-automated enactment of pragmatic-reuse plans

Reid Holmes and Robert J. Walker

Laboratory for Software Modification Research
Department of Computer Science
University of Calgary
Calgary Alberta, Canada

Abstract. Reusing source code in a manner for which it has not been designed (which we term a *pragmatic-reuse task*) is traditionally regarded as poor practice. The unsystematic nature of these tasks increases the likelihood of a developer pursuing one that is infeasible or choosing not to pursue a feasible one. In previous work, we demonstrated that these risks can be mitigated by providing support to developers to help them systematically investigate and plan pragmatic-reuse tasks. But planning is only a small part of performing a pragmatic-reuse task; to enact a plan, the developer would have to manually extract the code they want to reuse and resolve any errors that arise from removing it from its originating system. This paper describes an approach that semi-automates the process of pragmatic-reuse plan enactment, automatically extracting the reused source code and resolving the majority of compilation errors for the developer through *lightweight* (i.e., computationally simple but analytically unsound) transformations. By reducing the number of low-level compilation issues (which are typically trivial but copious) that the developer must resolve, they are able to focus on the higher-level semantic and conceptual issues that are the main barrier to the successful completion of the reuse task. The efficacy of our approach to save developer effort is evaluated in a small-scale, controlled experiment on non-trivial pragmatic-reuse tasks. We find that our approach improves the likelihood of a pragmatic reuse task being successful, and decreases the time required to complete these tasks, as compared to a manual enactment approach.

Keywords: Pragmatic software reuse, lightweight source code transformations

1 Introduction

As developers write code, they encounter situations where the functionality they are developing is familiar to them; either they have developed something similar before, or they know of some existing software that provides similar functionality [1]. Unfortunately, the existing functionality is often not designed in a way that permits its reuse in a black-box manner (e.g., as a framework, component, or product line) [2]. The developer is then left with few choices: re-implement the functionality, which is expensive and does not leverage existing mature code; re-modularize the existing code, which can be expensive and may not make sense for the original system; or to reuse the existing code in an ad hoc copy-and-modify process, which can lead to poor decisions being made [3, 4]. Copy-and-modify is often the *pragmatic* choice for software reuse in real scenarios.

Industrial developers undertake pragmatic-reuse tasks as a means to save time and leverage the testing effort put into existing source code; however, traditional approaches to pragmatic reuse can lead the developer to make poor decisions: they can commit too early to completing infeasible tasks; or they can avoid feasible reuse tasks due solely to superficial complexities. In previous research we addressed the shortcomings of traditional pragmatic-reuse approaches by supporting a lightweight means for simultaneously investigating and planning pragmatic-reuse tasks [5]. While planning can greatly increase the developer's comprehension of a pragmatic-reuse task, a plan remains an abstract artifact; for a plan to be useful, it must be followed ("enacted") to successfully complete the pragmatic-reuse task. Without tool support to enact pragmatic-reuse plans, three shortcomings remain: (1) the developer has to manually locate and integrate reused source code; (2) the abundance of low-level compiler errors during integration can obscure more complex high-level semantic and conceptual issues; and (3) investigating different options in the reuse plan can be expensive due to the costs of repeatedly manually modifying the source code.

Previous research has examined means for automatically or semi-automatically making adaptive modifications to source code. Much research requires that the source being adapted must be designed for reuse [6–9] or at least be compilable before adaptation [10, 11]; these requirements are not met in pragmatic-reuse scenarios. Some work has considered means for automatic adaptation, but requires that the adapted entities be formally specified [12, 13]; such formal specifications are not typically available in pragmatic-reuse contexts.

While pragmatic-reuse plans can help the developer to understand the reuse task at a high-level, it removes him from the realities of the source code; without enacting the plan it is difficult to tell if a decision in the plan makes the reuse task infeasible. To reduce the effort needed to enact a pragmatic-reuse plan, we have designed an approach for the semi-automatic enactment of these plans. Key to our approach is the application of lightweight (i.e., computationally simple but analytically unsound) transformations to the reused source code. We have implemented this approach in a tool called Procrustes[1], a plug-in to the Eclipse integrated development environment (IDE)[2]. Procrustes copies the code to be reused to the developer's project, and then modifies the code to be reused to minimize the number of dangling references the developer must inspect and correct; dangling references that cannot be transformed easily are flagged for the developer's attention. By semi-automating the enactment phase, the developer can instantly receive feedback on the implications of the plan and focus on the higher-level semantic problems that may inhibit the reuse task. This feedback loop makes it practical to quickly consider alternative decisions for a particular reuse task, thereby allowing the developer to create a higher-quality result.

We have performed an initial evaluation of our approach through two investigations into pragmatic-reuse tasks on two medium- to large-scale systems: a comparative case study to determine the minimum necessary effort for these tasks; and a controlled ex-

[1] The name comes from a figure in Greek mythology who would invite passersby to lie in a bed, whereupon he would force them to fit by stretching or amputation.

[2] http://eclipse.org (v3.2.1)

periment involving 8 developers enacting pragmatic-reuse plans manually and with our semi-automated approach.

The remainder of the paper is structured as follows. Section 2 provides additional background about pragmatic-reuse plans. Section 3 describes our lightweight approach for semi-automating the enactment of pragmatic-reuse plans. Related work is considered in Section 4. Our evaluation is presented in Section 5. Remaining issues are discussed in Section 6.

Our previous work contributed a method for investigating and planning pragmatic-reuse tasks. In contrast, this paper contributes a lightweight approach for the semi-automatic enactment of pragmatic-reuse plans; this is evaluated by comparing the success rates of pragmatic-reuse tasks with and without our lightweight approach.

2 Background: Pragmatic-Reuse Plans

Developers currently perform pragmatic reuse tasks manually. They identify some fragment of source code they want to reuse and integrate with their project. They then successively traverse through compiliation errors that have arisen and resolve them one at a time (integrating more code as necessary). Unfortunately, for large reuse tasks it is difficult to tell at the outset if the task will be successful. We introduced the concept of the pragmatic-reuse plan to help developers understand the scope of pragmatic-reuse tasks before the investment of integrating the reused code [5].

A pragmatic-reuse plan consists of a list of tagged program elements. Classes, methods, and fields can be marked as as **accepted** ("I want to reuse this code"), **rejected** ("I don't want to reuse this code"), or **remapped** ("I want to redirect dependencies on this code to be on something within my own system"). Special cases exist for **injecting** code into classes and **extracting** fields by reusing only specific fields from a class.

3 Procrustes: Semi-automating Enactment

Procrustes bridges the gap between the conceptual intent of a pragmatic-reuse plan and the realization of the task. To do this, Procrustes copies the source code fragments that the developer intends to reuse from the originating project into the target project (see Section 3.1) according to the pragmatic-reuse plan. Using the plan, Procrustes integrates the reused code with the target project by resolving the dangling references that arose from removing the code from its originating environment (see Section 3.2).

3.1 Extraction

After the developer activates Procrustes (by pressing the "enact plan" button in the IDE), it locates all of the source code corresponding to accepted nodes in the reuse plan and first copies this code into the target project. The original package hierarchy is maintained within the extracted code, for ease of comprehension.

After Procrustes has copied the code into the target project, dependencies between the reused classes will remain valid as the package structure was maintained. Any dependencies to structural entities outside those being reused would normally cause compilation problems; however, the integration portion of Procrustes resolves many of these (see Section 3.2).

3.2 Integration

During this phase, the source code that has been migrated from the original system to the target system must be manipulated to resolve any compilation problems that have arisen. When source code is removed from the context for which it was written and placed into a foreign environment, many of its dependencies can be unfulfilled. The unfulfilled dependencies in the reused code are manifested as dangling references to classes, methods, and fields that were not reused (and do not exist in the target project).

Using the model that represents the pragmatic-reuse plan, Procrustes can precompute each of the changes that the tool should perform to repair many dangling references. The integration process proceeds in four main steps:

Managing source code additions. The code addition step adds new code to those entities previously migrated to the target project in Section 3.1. There are two cases that must be handled: code injection, and field extraction. For code injection, any fragment provided by the developer is inserted into its target class (as specified in the reuse plan).

For field extraction, any fields in the plan that have been marked for extraction are copied from the class in which they were declared into the target class specified in the pragmatic-reuse plan. Moving the field only updates its declaration, not its references in the reused code (this happens in the next step). Again, the import statements are also updated to reflect this addition to the target class.

Managing dangling references. The management of dangling references is the most complex step in the integration phase. Two primary classifications of dangling references are managed: (1) references to fields, calls to methods, and references to supertypes that were rejected in the reuse plan; and (2) calls to methods and referenced fields that have been injected, extracted, or remapped in the reuse plan.

Procrustes searches each accepted source element for dependencies on other elements that have been rejected. If a dependency on a rejected field or method is found, it is managed by commenting-out the entire statement containing code corresponding to the dependency within the accepted code. Procrustes comments-out rejected dependencies, rather than remove them completely, rather than remove them completely, as their details could still be informative to the developer. These comments are accompanied with a tag to indicate that the change was made by Procrustes. This also allows the developer to easily locate each change to the source made by Procrustes using traditional search tools. For the sake of simplicity, Procrustes rejects field references and method calls only at the statement level; despite the inherently unsound nature of such a lightweight approach to transformations, in practice we have found it to be effective (Section 5).

If the pragmatic-reuse plan has reused a class but not some subset of its supertypes, the tool must remove references to those supertypes. This often occurs as developers trim the functionality they are interested in from an inheritance hierarchy. Any number of supertypes can be removed. If the subclass was dependent on a method within a rejected supertype this would be shown as a dependency between a method in the subtype and a method on the supertype during the planning process. This dependency would have been resolved at the beginning of this step.

Finally, any accepted element with a structural dependency that has been remapped is handled. These cases are simpler than in the rejected-element case as code does not disappear; it is simply redirected. This step handles 5 cases: calls to injected and remapped methods and references to injected, extracted, and remapped fields.

Managing unnecessary code. This step removes methods and fields marked as rejected in the reuse plan that are declared within accepted classes and interfaces. For code readability, rejected fields and methods are completely removed from the source code by removing them from their containing class, rather than just being commented out. The tool only needs to remove methods and fields that are children of classes that have been accepted, i.e., if a type is completely rejected or remapped Procrustes does not need to delete any code as it would not have been integrated to the target system.

Finalizing source code modifications. Each of the changes made by the three previous steps were made to an intermediate representation of the code, not directly to its text. This separation minimizes the chances that one change will cause another alteration to fail. After all the steps are complete, Procrustes applies the changes to the files and writes them to disk, collecting statistics about the scope of the changes it has made.

3.3 Implementation

Procrustes is implemented as an Eclipse plug-in. By creating the tool as a plug-in we are able to leverage many of the features that Eclipse provides for parsing, compiling, and manipulating Java files. Each of the nodes in the reuse plan matches a specific structural element in Eclipse's Java abstract syntax tree (AST). All of our changes are recorded using the Eclipse `ASTRewrite` class. This class aggregates the changes made during each of the steps of the integration phase; the source files are all changed and written to disk only at the end of this phase.

4 Related Work

Previous work in a variety of areas bears similarity to the problem we address; however, none meets all the requirements for (semi-) automating the enactment of pragmatic-reuse plans.

Most reuse literature emphasizes designing for reuse, for example, in object-oriented programming [14], frame-based reuse [7], domain-specific language-based approaches [6], and component-based approaches [8, 9]. Such approaches are inappropriate in our context, as pragmatic reuse entails situations where the original design did not anticipate the desired reuse scenario.

Transformation-based approaches to reuse were prevalent in the 1980s, for example, that of Feather [10]. Such approaches were based on the notion of formally correct refinement, thus requiring compilable programs and (usually) formal specifications; neither is available in our context. Jackson and Rinard recognize the continuing value of unsound analyses [15], for both their usefulness and ease of efficient implementation.

Several systems have been developed to identify reusable components. Lanubile and Visaggio developed a technique based on program slicing to identify reusable source code [16]. The CARE system [17] identifies and extracts reusable components using a

metrics-based approach; the applicable components must have no or few external dependencies. These systems do not allow the developer to specify which detailed entities are to be reused, or how to deal with problematic dependencies.

Various approaches attempt to adapt code for use in a novel context. The Adapter object-oriented design pattern [11] adapts classes or objects to conform to a required interface, but maintains all the dependencies of the original classes or objects; in our context, we need to be able to eliminate or replace such dependencies. Approaches like that of Gouda and Herman [12] and of Yellin and Strom [13] automatically adapt components to new contexts; however, they require complete, formal specifications to operate that are not typically available or appropriate for pragmatic-reuse tasks.

5 Evaluation

The intent of our approach is to greatly reduce the effort required to enact a pragmatic-reuse plan. By reducing this effort, developers can better judge the merits of their reuse tasks to maximize their chances of having desirable outcomes. To evaluate Procrustes we set out to answer two questions: (1) How much effort can semi-automating the enactment of pragmatic-reuse plans save developers? (2) Does semi-automating the enactment of pragmatic-reuse plans affect the outcomes of pragmatic-reuse tasks performed by developers? Each question was addressed with its own evaluation.

In both evaluations, "completion" was defined as the successful execution of a test suite that we provided for the purpose. One test suite was implemented as an Eclipse plug-in, while the other was a standalone application; we henceforth refer to both as test harnesses.

5.1 Task descriptions

Both evaluations used the same two tasks. Each of these tasks involved extracting specific functionality from an existing system and integrating it into a new system. Each task operated on a different open-source Java system from a different domain.

Metrics lines-of-code calculator. The Metrics Eclipse plug-in[3] can compute 23 different metrics (e.g., lines of code, cyclomatic complexity, efferent coupling, etc.) for resources inside Eclipse projects. This plug-in contains 229 classes comprising 14.5 thousand lines-of-code (kLOC). The goal of this task was to reuse the lines-of-code (LOC) calculator from this project; however, the system was not designed to enable individual metrics to be reused without the remainder of the Metrics plug-in.

The reuse plan for this task involved reusing portions of 8 classes. Successful completion of this task involved reusing 392 LOC. For the task to be a success, the reused code had to compute the LOC for every class in every project in the Eclipse workspace when activated by the test harness.

Azureus network throughput view. Azureus[4] is a client application for the BitTorrent peer-to-peer file-sharing protocol. Azureus contains 2,257 classes comprising 221 kLOC. It contains a view that visualizes its network throughput for the user. The

[3] http://metrics.sf.net v1.3.6

[4] http://azureus.sf.net v2.4.0.1

goal of this task is to extract this network throughput view from Azureus and integrate it into a new system. This feature was not designed to be reused outside of Azureus.

The reuse plan for this task involved reusing portions of 6 classes. Successful completion of this task involved reusing 366 LOC. To succeed at the task, the reused code had to be able to correctly display a data set provided in the test harness.

5.2 Analysis of minimum required effort

Our first evaluation considered how much effort Procrustes can save a developer by semi-automating the enactment of a pragmatic-reuse plan. We performed both tasks both manually and using Procrustes. The number of compilation errors present after copying all of the required code (in the manual case) or pressing the "enact plan" button in the Procrustes-supported case is given in Table 1. These numbers are the first indicator to the developer of the amount of work facing them before they can complete the task.

Case	Procrustes	Manual	Error reduction
Metrics	3	62	95.2%
Azureus	11	32	65.6%

Table 1. Compilation errors for each task and treatment.

Compilation errors alone are not always a good indicator of required effort. Often, making one small change in the source code can remedy (or create) several errors. To get a true sense of the minimum amount of effort the developer would need to expend to successfully complete each reuse task, we investigated each task in terms of "edits". An edit represents a single conceptual change the developer makes to the source code. The minimum number of edits required to successfully complete each task with each treatment is given in Table 2.

Case	Procrustes	Manual	Edit reduction
Metrics	2	60	96.7%
Azureus	4	25	84.0%

Table 2. "Edits" required for each task and treatment.

Some edits require more thought and investigation on the part of the developer to resolve than others. These difficult edits arise due to conceptual mismatches between the original and target systems [2]. For the Metrics task, only one of the edits represented a conceptual mismatch that arose from removing the reused code from the system for which it was designed. Three edits in the Azureus task represented conceptual mismatches; these were common between the two treatments. While Procrustes does not resolve any of the conceptual mismatch errors, it helps the developer to quickly identify them by resolving all of the trivial compilation errors that occlude them. This difference is especially evident when a developer repeatedly iterates on a reuse plan.

5.3 Task effectiveness experiment

Our second investigation sought to determine if developers performing pragmatic-reuse tasks had better outcomes using Procrustes. We performed a controlled experiment

with eight developers. Four of these were industrial developers (I1 through I4) and four were software engineering graduate students (G1 through G4). The participants had between 6 years (an industrial developer) and 12 years of experience (also an industrial developer). Each participant was randomly assigned a task–treatment pairing. Each task–treatment pairing was completed by two graduate students and two industrial developers. Each participant used Procrustes for one task and performed the other task manually. We created a reuse plan for each task and provided identical versions for each treatment. A time limit of one hour was set for each task; we chose this time limit as it seemed like a reasonable amount of time for a developer to invest in this kind of task. We recorded whether or not the participants succeeded or failed for each task, how long they spent performing the task, and collected their final code for later analysis. After completing both tasks the participants completed a follow-up questionnaire (see the website cited earlier for details).

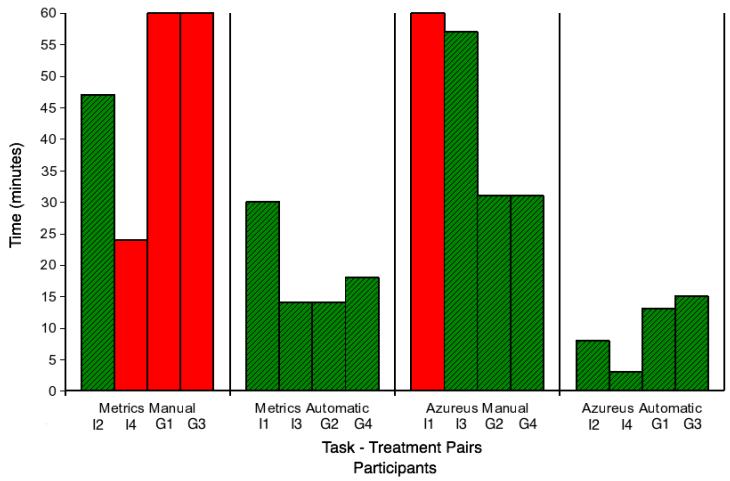

Fig. 1. Results of the task effectiveness experiment. Green/hatched bars indicate success. Red/solid bars indicate failure.

The results of the experiment are shown in Figure 1. The figure depicts successful task–treatment pairings in green (diagonal hatching in greyscale). Those task–treatment pairings that were failures are indicated in red (solid in greyscale). The graph clearly shows that the participants successfully completed more tasks using Procrustes (8 out of 8) than with the manual treatment (4 out of 8). It is also clear that, for these tasks, developers were able to complete the tasks in less time using Procrustes than when undertaking them manually.

Manual treatment. The four participants who manually enacted the Metrics LOC reuse task were the least successful. By examining their resulting code and reading their comments in the questionnaire, it became clear that they knew they had a problem to fix, but they did not know where this problem was. At the outset of this manual task, each participant (I2, I4, G1, G3) had 62 compilation errors to resolve; in the process of

resolving these errors, they seemed to become disoriented. While each of them ended up with code that compiled, only I2 successfully completed the task (in 47 minutes). One of the participants became so frustrated with this task that after 24 minutes he gave up. One of the participants who failed at this task (I4) reported, "The manual approach was mostly drone work; it took longer to get the target project into a state where interesting problems could be solved." Even I2, who was successful, stated, "[The manual task was] not hard, very tedious though. I was sitting there going 'this should be automated.'"

The other 4 participants undertook the manual version of the Azureus task. Only one of these developers (I1) failed to complete the task (after 60 minutes); the rest managed to finish in an average of 40 minutes.

Automated treatment. Each of the participants who undertook the Metrics LOC task using Procrustes managed to successfully complete the task (in an average of 19 minutes). In the questionnaires, these developers mentioned that they were able to concentrate on the 3 compilation errors that remained after Procrustes ran. Since two of these errors were trivial, they were able to focus on the single remaining error (which was a conceptual mismatch). While this error was tricky to solve, each of them was able to get the code to work successfully with the test harness.

All the participants also completed the Azureus task using Procrustes (in an average of 10 minutes). These developers did not seem to have any trouble changing the reused code to use the specific fields for blue rather than the `blues` array. For this task, I4 said, "There where still some syntactic mismatches, but what I found was that the problems that remained were more directly related to the actual misalignments between use contexts; they were more directly related to the reuse I was trying to achieve."

5.4 Lessons learned

The first evaluation demonstrated the amount effort that Procrustes can save the developer. The second evaluation showed that this savings can translate into an increased rate of success for pragmatic-reuse tasks. The controlled experiment showed that Procrustes enabled the developers to quickly locate the conceptual mismatches that were the real barrier to completing the tasks; these mismatches could not be found by looking at the reuse plan alone.

6 Discussion

In this section, we consider a number of issues regarding our claims, evidence, and conjectures about the current work, and where the work should go from here.

6.1 Does semi-automating enactment matter?

A pragmatic-reuse plan remains an idealization until it is enacted to complete a pragmatic-reuse task. Semi-automation aids the developer by eliminating the most trivial issues so that the developer can focus on addressing non-trivial problems. In contrast, Participant G1 failed at the manual treatment of the Metrics task because he could not resolve the conceptual mismatch between the source and target environments; he reported spending 40 of 60 minutes resolving low-level compilation errors. This pattern

recurred: in all 4 of the unsuccessful manual treatments, the developers were able to get the code to compile without error; however, in 3 of the 4, the developers also used their full 60 minutes without completing the task.

We believe this time-savings also matters from an industrial perspective; developers are only willing to invest a limited amount of time into these kinds of tasks before building the functionality from scratch. As Participant I2 states, "I wouldn't use [a pragmatic-reuse plan] on its own... I don't care that the model is nice, it's running code that counts. The automation provided by Procrustes gives me the bridge I need to make the model valuable." And, "[The manual approach] gives me a jigsaw puzzle where I've opened the box and have all the pieces. [Procrustes] gives me a 90% complete puzzle and I just have to put in the last pieces. It's what makes the tool usable to me."

6.2 Do lightweight transformations suffice?

We have developed Procrustes specifically to minimize the number of compilation errors associated with reusing source code that was not designed for reuse. As such, it applies only the most basic of transformations. Significant conceptual mismatches [2] will not be repaired by these transformations. However, in our experience we have found that the vast majority of mismatches encountered are trivial in nature, but copious, and that the transformations applied by Procrustes suffice to repair these.

As Participant I4 states, "The automatic enactment brought the target project into a state where I could more immediately start working out the higher-level mismatches between contexts. The remaining compilation errors were more directly related to these [higher-level] mismatches. In contrast, the manual enactment involved a lot more iterations of compile/fix and the errors [were] more low-level."

Using a more complex means of specifying the transformations could allow the enactment task to be fully automated. We suspect that the cost of using a fully-expressive transformation language would be significantly greater than the cost of manual enactment. Procrustes provides an alternative to these two approaches.

6.3 Representativeness of participants and tasks

The number of participants was fairly small, at only eight. We have not attempted to quantify the relative effort of the treatments with statistical significance and have reported times only to give a sense of scale and trends. Half of our participants were industrial developers and half were experienced graduate students. While there was some variation between these two groups, the trend in the results is unambiguously in favour of Procrustes.

Our first evaluation also considered the "ideal" developer who could complete her tasks with the minimum amount of work. While this developer is also not representative, she does represent the lower-bound that the best developers could strive to achieve. Even this developer had to perform considerably more work with the manual treatment than with the semi-automated one.

Only two tasks were performed on two systems. Each of these tasks were non-trivial, being taken from real development scenarios and not synthesized for the sake of the research; each involved the reuse of functionality that had not been designed to

be reused in the way we needed. The systems were of medium- to large-scale from two disparate domains.

For the sake of experimental control, we provided the participants with pragmatic-reuse plans for their tasks. While this control enabled us to compare the effectiveness of multiple developers performing the same tasks with different treatments, future evaluations will involve developers creating and enacting their own reuse plans.

6.4 Net cost of pragmatic reuse

One might question how much effort must be expended to create or to interpret pragmatic-reuse plans, and whether this effort would overwhelm the reported benefits of semi-automated enactment. In our experiment, each pragmatic-reuse plan required less than 30 minutes to construct by us, despite our lack of experience with the original systems. The 30 minute investment required to create the plans involved gaining an understanding of the originating system, something that developers would need to do in both manual and tool-supported scenarios.

While our evaluation indicates that Procrustes can help with pragmatic-reuse tasks, going forward we will have to evaluate the relative performance of three different cases: (1) a developer creating functionality from scratch; (2) a developer manually reusing similar functionality from a pre-existing system; and (3) a developer creating a pragmatic-reuse plan, and enacting it with Procrustes. With such an evaluation, we would like to gain an insight into the relative merits of tool-supported end-to-end pragmatic-reuse scenarios compared to more traditional unanticipated-reuse approaches.

7 Conclusion

This paper has described Procrustes, a tool for semi-automatically enacting pragmatic-reuse plans. By resolving the bulk of the compilation errors that arise from reusing a piece of source code out of the context it was designed for, Procrustes allows the developer to focus their effort on resolving errors that represent conceptual mismatch between the source and target systems.

We performed two evaluations to determine the efficacy of Procrustes. In the first evaluation we found that a putative developer doing the least amount of work possible would have to perform significantly more work to enact a pragmatic-reuse plan manually than with Procrustes. The second evaluation found that participants using Procrustes were far more likely to successfully complete a pragmatic-reuse task (8 out of 8 cases) than those performing the same tasks manually (4 out of 8 cases). Additionally, in all successful tasks, the use of Procrustes reduced the time needed for the enactment process to between 25% and 40% of the times for the manual treatments, on average. The implication of these studies is that semi-automation can make it feasible for the developer to iterate on their reuse plan, allowing them to explore the concrete realization of different plan alternatives, leading to higher-quality reuse plans and more successful reuse tasks.

Procrustes utilizes computationally simple but analytically unsound transformations to enact a pragmatic-reuse plan. Because of their simplicity, these transformations are fast to perform. Despite their lack of soundness, they can effectively filter out trivial

mismatches from the developer's attention, allowing them to focus on whether and how to address non-trivial mismatches. The lightweight nature of our approach, coupled with our appreciation of the needs of the developer, are key to its success.

8 Acknowledgments

We wish to thank Brad Cossette, Rylan Cottrell, Jonathan Sillito, and the other members of the Laboratory for Software Modification Research for their helpful comments. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and in part by IBM Canada.

References

1. Sen, A.: The role of opportunism in the software design reuse process. IEEE Transactions on Software Engineering **23**(7) (1997) 418–436
2. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. IEEE Software **12**(6) (1995) 17–26
3. Garnett, E.S., Mariani, J.A.: Software reclamation. Software Engineering Journal **5**(3) (1990) 185–191
4. Krueger, C.W.: Software reuse. ACM Computing Surveys **24**(2) (1992) 131–183
5. Holmes, R., Walker, R.J.: Supporting the investigation and planning of pragmatic reuse tasks. In: Proceedings of the International Conference on Software Engineering. (2007) 447–457
6. Neighbors, J.M.: Draco: A method for engineering reusable software systems. In Biggerstaff, T.J., Perlis, A.J., eds.: Software Reusability. Volume 1: Concepts and Models of ACM Press Frontier. Addison–Wesley, Boston, United States (1989) 295–319
7. Bassett, P.G.: The theory and practice of adaptive reuse. In: Proceedings of the Symposium on Software Reusability. (1997) 2–9
8. Mezini, M., Ostermann, K.: Integrating independent components with on-demand remodularization. In: Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2002) 52–67
9. Estublier, J., Vega, G.: Reuse and variability in large software applications. In: Proceedings of the Foundations of Software Engineering. (2005) 316–325
10. Feather, M.S.: Reuse in the context of a transformation-based methodology. In Biggerstaff, T.J., Perlis, A.J., eds.: Software Reusability. Volume 1: Concepts and Models. Addison–Wesley (1989) 337–359
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison–Wesley (1994) Chapter on the Adapter design pattern.
12. Gouda, M.G., Herman, T.: Adaptive programming. IEEE Transactions on Software Engineering **17**(9) (1991) 911–921
13. Yellin, D.M., Strom, R.E.: Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems **19**(2) (1997) 292–333
14. Johnson, R.E., Foote, B.: Designing reuseable [*sic*] classes. Journal of Object-Oriented Programming **1**(2) (1988) 22–35
15. Jackson, D., Rinard, M.: Software analysis: a roadmap. In: Proceedings of the Conference on The Future of Software Engineering. (2000) 133–145
16. Lanubile, F., Visaggio, G.: Extracting reusable functions by flow graph-based program slicing. IEEE Transactions on Software Engineering **23**(4) (1997) 246–259
17. Caldiera, G., Basili, V.R.: Identifying and qualifying reusable software components. Computer **24**(2) (1991) 61–70