Program Metamorphosis

Christoph Reichenbach*, Devin Coughlin, and Amer Diwan

University of Colorado, Boulder, CO 80309, USA
{reichenb,devin.coughlin,diwan}@colorado.edu

Abstract. Modern development environments support refactoring by providing atomically behaviour-preserving transformations. While useful, these transformations are limited in three ways: (i) atomicity forces transformations to be complex and opaque, (ii) the behaviour preservation requirement disallows deliberate behaviour evolution, and (iii) atomicity limits code reuse opportunities for refactoring implementers.

We present 'program metamorphosis', a novel approach for program evolution and refactoring that addresses the above limitations by breaking refactorings into smaller steps that need not preserve behaviour individually. Instead, we ensure that sequences of transformations preserve behaviour together, and simultaneously permit selective behavioural change.

To evaluate program metamorphosis, we have implemented a prototype plugin for Eclipse. Our analysis and experiments show that (1) our plugin provides correctness guarantees on par with those of Eclipse's own refactorings, (2) both our plugin and our approach address the aforementioned limitations, and (3) our approach fully subsumes traditional refactoring.

Keywords: Refactoring, Program Evolution.

1 Motivation

Modern programming methodologies, such as Extreme Programming [2], use *refactoring* to prepare software for impending change or to eliminate "bad smells" in source code. Fowler *et al.* [4] defines refactoring as:

> *A change made to the internal structure of software ... without changing its observable behaviour*

To automate this process, integrated development environments such as Eclipse [14] and refactoring engines such as HaRe [8] provide machine support for refactoring. These systems implement refactoring as atomic transformations guarded by preconditions. The underlying assumption is that if the precondition holds, the transformation will preserve behaviour. If the precondition does not hold, the IDE disallows the transformation. This approach prevents some forms of unintended behavioural change but presents several problems to refactoring users and developers:

* Supported by NSF Career Grant CCR-0133457 and NSF Grant ST-CRTS 0540997.

S. Drossopoulou (Ed.): ECOOP 2009, LNCS 5653, pp. 394–418, 2009.
© Springer-Verlag Berlin Heidelberg 2009

```
class A {                           class A { }
  private int x;  ┐
  void f()        ├─┐               class B {
  { x = C.g(); }  ┘ └──>  {           private int x;
}                                     void f()
                                      { x = C.g(); }
class B { }                         }
```

Fig. 1. The chicken-or-egg problem: fields and methods must be moved simultaneously

1. **The chicken-or-egg problem.** Atomic transformations with preconditions may prohibit safe refactorings. Consider the program in Figure 1: if we wish to move field “x” and method “f” from class “A” to class “B”, we would like to employ the ‘Move Method’ and ‘Move Field’ refactorings [4]. If we move “f” first, “B.f()” will not be able to see “x”, so ‘Move Method’ will disallow the move. However, if we move “x” first, the process fails for the converse reason. This chicken-or-egg problem is exacerbated if additional fields or mutually recursive methods are affected, and while refactoring users can sometimes find workarounds, they may find it easier to abandon the promised behaviour preservation of refactorings in favour of faster manual editing. Some refactoring implementations address this problem by attempting to predict which additional methods and fields must be moved simultaneously to atomically perform the refactoring, but these fixup heuristics are complex, error-prone, and may run contrary to the user’s wishes.
2. **The selective behaviour evolution problem.** The user may want to exploit the automation provided by refactorings without necessarily preserving all behaviour. For example, in theory a refactoring must never allow the user to rename a public method since some independent source code (perhaps in a plugin) might reference this method by name. In practice, a user might accept this change and yet still want to prevent other forms of behavioural change, such as a renamed method overriding a method it didn’t override before. This is a dilemma for refactoring engine designers: they must anticipate the degree to which users value safety over versatility.
3. **The predictive analysis problem.** Since traditional behaviour preservation checks are implemented as preconditions, they must predict the effect of the transformation in order to determine if it will cause problems. To do this exhaustively is quite difficult. For example, in the case of ‘Rename’, a precondition must consider all the possible ways in which a name could be captured and check to see if that will happen. As Schäfer *et al.* [13] point out, a less error-prone approach is to first perform the transformation and then check after the fact to see if any names have been captured.

This paper shows how a small twist to the “classical” refactoring implementation strategy allows us to solve the above problems. We achieve this by:

1. Capturing the approximate program behaviour in a program model,
2. Applying a series of possibly non–behaviour-preserving program metamorphosis steps (PM steps for short),
3. Using *postconditions* to compare the original program model to the current program model to see if behaviour has changed.

This twist yields a new view on refactoring: instead of treating refactoring as the application of atomic refactorings that must preserve behaviour by themselves, we can think of refactoring as a *process* of gradual application of PM steps. As part of this process, users may check for behaviour preservation at any time. Thus, users may decide to first transform their program as they desire (e.g., moving fields and methods) and then either recover behavioural equivalence, if necessary, or expressly and selectively accept some or all behavioural change.

Program metamorphosis provides three main benefits over the traditional approach to refactorings:

1. It allows safe transformations through intermediate stages whose behaviour differs from the intended behaviour,
2. It allows safe transformations through intermediate stages that may not even compile,
3. It allows the user to selectively evolve behaviour.

In this paper, we provide the following contributions: we describe the process of program metamorphosis (Section 2) and demonstrate the benefits it offers over traditional refactoring (Section 3). We then sketch a theory that allows us to view program metamorphosis as a decomposition of refactorings and show that our approach is at least as safe as traditional refactoring, if we base it on refactorings that we can decompose in a certain way (Section 4). Next, we describe a prototype Eclipse plugin that demonstrates that program metamorphosis is practical to implement (Section 5). Finally, we describe several refactorings we implemented using our prototype; where possible, we compare the quality of their behaviour preservation promises against those provided by Eclipse's refactorings by applying both to Java projects with comprehensive unit test suites (Section 6). Section 7 reviews related work and Section 8 concludes.

2 The Process of Program Metamorphosis

Before we look at concrete examples of program metamorphosis, it is helpful to consider the structure of the underlying process. A program metamorphosis system consists of three main components:

1. a mechanism for generating *program models* that describe the approximate behaviour of a program,
2. a *consistency checker* that compares two program models and extracts the inconsistencies (if any) between them,
3. a suite of small *PM steps*, each of which transforms the program in a possibly non–behaviour-preserving way.

Program metamorphosis uses the consistency checker and program model generator to help the user compose PM steps into a sequence of transformations that, taken as a whole, preserves behaviour.

2.1 Combining Multiple PM Steps

Unlike traditional refactorings, which use preconditions to check the legality of the transformation, program metamorphosis uses postconditions. Specifically, program metamorphosis constructs a model of the program's behaviour before the transformation, transforms the program, and then constructs another model after the transformation, possibly re-using parts of the earlier model. If the two models do not match, then the program's behaviour may have been changed. This approach, illustrated in Figure 2, allows program metamorphosis to safely combine multiple transformations as follows:

1. We first calculate a model for the program and save it as the "desired program model." We call this the "desired" model since we ultimately want the transformed program to end up with the same model.
2. When the user applies a PM step, we compare the desired program model with the calculated current program model, reporting any inconsistencies to the user. If there are inconsistencies then the user may:
   (a) Revert the previous step,
   (b) Apply another PM step, or
   (c) Accept any or all of the reported inconsistencies as behavioural change by updating the desired model to incorporate the change in behaviour.

The key benefit of program metamorphosis is that after the user has applied any particular PM step, the program's current behaviour may not match its original behaviour, but *the user can continue applying steps until it does.* In this way, PM allows the composition of simple, possibly non-behaviour preserving steps into a sequence that does, in its entirety, preserve behaviour.

Figure 3 visualises the advantages of our approach. In this figure, every vertex represents a program; either well-formed (black) or ill-formed (white), while edges represent PM steps. Programs with 'equivalent' behaviour are grouped into equivalent classes. To refactor program p_a to program p_b, we can choose

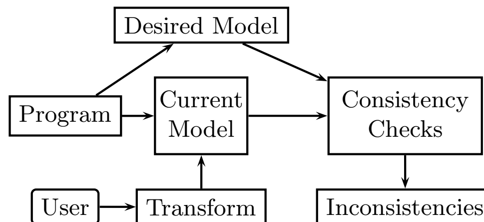

Fig. 2. Consistency checking process for program metamorphosis

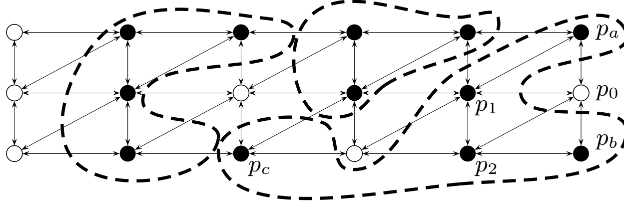

Fig. 3. Program metamorphosis. Black vertices represent well-formed programs; white vertices represent ill-formed programs. Solid edges represent the application of PM steps. Dashed lines group programs with equivalent behaviour.

two PM steps: from p_a to p_0, then to p_b, since we may 'pass through' ill-formed programs. With refactoring, we must always remain well-formed and in the same equivalence class, so we have to take a longer route through p_1 and p_2 instead (Section 3.1 gives a concrete example of this scenario). Worse, we can never hope to reach p_c from p_a with refactoring, while there are many ways to get there with program metamorphosis (Section 5.3 gives a concrete example of this scenario).

2.2 Recovery Plans

In addition to relying on the user to apply additional steps when the current model does not match the desired model, a program metamorphosis system can automatically attempt to recover consistency in several ways:

1. *Disallow/retract PM steps* that fail to preserve behaviour.
2. *Heuristically apply supporting PM steps.* For example, Schäfer *et al.* [13] describe a particular technique that can be used to automatically fix name capture after 'Rename'. In some cases, heuristic changes may have undesired side-effects; if so, the user must undo them later. In a similar vein, existing refactoring tools, such as Eclipse, predict conflicts that will happen and heuristically pre-apply other refactorings in order to avoid them.
3. *Search for recovery plans.* Recovery plans are short sequences of PM steps that will satisfy the postconditions. The user can then pick which plan (if any) she wants to enact. This approach falls within the realm of *AI Planning*; to be practical, it requires heuristics to guide the planning process.
   One of our earlier prototypes incorporated such facilities, though our initial experiments suggested that scaling this approach is nontrivial. We expect to explore this idea further in future work.

2.3 Challenges in Comparing Program Models

Program equivalence is undecidable; thus, we cannot be fully precise when comparing two program models. We can choose to err either on the side of being *pessimistic*, i.e., making conservative worst-case assumptions, or on the side of being *optimistic*. Both approaches have their merits: being pessimistic means

that we will always be safe, while being optimistic means that we can be more flexible. In Java we must be optimistic, at least in some respects; otherwise dynamic class loading and reflection render most interesting refactorings impossible.

Traditional refactorings have only two options: they can be pessimistic and safe, or optimistic and flexible. Program metamorphosis adds a third option: it can be pessimistic and safe, but also allow users to accept inconsistencies as behavioural change and thereby also be flexible.

We show in Section 4 that at least in theory program metamorphosis is as safe as refactorings; we show in Section 6 that our first prototype is also as safe as refactorings.

3 Examples of Program Metamorphosis

In the following, we present three examples to illustrate our approach in practice. First, we illustrate transformations that temporarily change behaviour (Section 3.1). Next, we consider transformations that temporarily render the program ill-formed (Section 3.2). Finally, we examine the use of selective behaviour evolution (Section 3.3).

3.1 Transformations through Non-equivalent Programs

Consider the program below and assume that the user wants to swap the names of the totalValue instance variable and the total method parameter. In the following, we have labelled important declarations and variable references with $[\cdot]^n$.

```
class Receipt {
  [int totalValue]^1;
  void setTotal([int total]^2) {
    [totalValue]^3 = [total]^4;
  }
}
```

A rename refactoring using atomic preconditions would disallow starting the transformation by renaming either totalValue to total or total to totalValue because in both cases the parameter (2) would capture the left-hand side of the assignment (3), possibly changing the behaviour of the program. This transformation *could* be accomplished via refactorings, albeit awkwardly, by first renaming one of the variables to a temporary name, renaming the other to the first name, and then renaming the temporary name to the second name, but this work-around requires three steps rather than two and forces the user to plan ahead when refactoring.

PM can perform the transformation safely in two steps using postconditions. It starts by creating the following program model, which captures the binding of variable uses to declaration (here $n \rightarrow m$ indicates that the variable used at n refers to the declaration at m).

Desired Name Model	3 → 1, 4 → 2

The user can then apply a rename step to change the total field declaration (1) to be called totalValue. PM now computes the model for the transformed program and compares it to the desired model, reporting any inconsistencies to the user. In this case, the rename has caused the reference on the left-hand side of the assignment (3) to be captured: it previously referred to the the field (declaration 1) but now refers to the parameter (declaration 2).

```
class Receipt {
  [int totalValue]¹;
  void setTotal([int totalValue]²) {
    [totalValue]³ = [totalValue]⁴;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → 2, 4 → 2
Inconsistencies	3 captured by 2

The program's behaviour has now been changed: calling setTotal() will no longer update the totalValue field. To ensure behaviour preservation, the user can either revert the rename transformation, or apply another rename step to rename the left-hand side totalValue (3) to total.

PM steps have access to the desired program model, which helps them in transforming programs. In the case of rename, we use the mappings in the *desired model*, rather than the current model, to decide which occurrences of totalValue need to be changed. Thus renaming (3) to total updates both (3) and the field declaration (1), since (3) is mapped to (1) in the desired model, while leaving (2) unaffected:

```
class Receipt {
  [int total]¹;
  void setTotal([int totalValue]²) {
    [total]³ = [totalValue]⁴;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → 1, 4 → 2
Inconsistencies	None

The current model now matches the desired model. We have achieved the desired transformation safely and naturally in only two steps, which would be impossible with a traditional Rename refactoring since atomic preconditions prohibit transforming through an intermediate stage that does not preserve behaviour.

3.2 Transformations through Ill-Formed Programs

Sometimes it makes sense to temporarily transform into a program that will not even compile. Consider the case below where the user wants to move both the setTotal() method and the total field from the Receipt class to the Bill class.

Preconditions, as in traditional refactorings, would disallow first moving the method, since there is no total field in Bill, but would also prohibit first moving the total field, since that would leave behind an unresolved reference to that field in Receipt.

```
class Receipt {
  [int total]^1;
  void setTotal([int totalValue]^2) {
    [total]^3 = [totalValue]^4;
  }
}

public class Bill { }
```

Desired Name Model	3 → 1, 4 → 2

By using postconditions and a program model, program metamorphosis avoids this chicken-or-egg problem: the user can move either the field or the method first, and then move the other. Suppose she moves setTotal() first: this step will will result in an unknown name inconsistency and the compiler will complain that it can't resolve the reference to total (3).

```
class Receipt {
  [int total]^1;
}

class Bill {
  void setTotal([int totalValue]^2) {
    [total]^3 = [totalValue]^4;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → ?, 4 → 2
Inconsistencies	Unknown name 'total' at 3

The user can now apply a second move step to move the total field to Bill. After this step, the current model matches the desired model and the program is again well-formed.

```
class Receipt { }

class Bill {
  [int total]^1;
  void setTotal([int totalValue]^2) {
    [total]^3 = [totalValue]^4;
  }
}
```

Desired Name Model	3 → 1, 4 → 2
Current Name Model	3 → 1, 4 → 2
Inconsistencies	None

3.3 Selective Behaviour Evolution

Refactorings are not, by definition, permitted to change program behaviour, but sometimes this is desirable. Consider the case where the user wants to rename a public class: technically this should not be allowed since there may be independent code (such as a plugin) that relies on the existence of that class. If we care about preserving our public APIs, we can extend the program model to account for class visibility. Suppose, in the example below, that the user wants to change the Bill class to be called Invoice instead.

```
public class Bill {
  [private int total]^1;
  private void setTotal([int totalValue]^2) {
    [total]^3 = [totalValue]^4;
  }
}
```

Desired Visibility Model	public → {Bill}, protected → {}

After renaming Bill to Invoice, the current and desired models no longer match:

Desired Visibility Model	public → {Bill}, protected → {}
Current Visibility Model	public → {Invoice}, protected → {}
Inconsistencies	Extra public class Invoice Missing public class Bill

Program metamorphosis now reports two inconsistencies between the current and desired program models. First, there is a new public class, Invoice, whose presence could prevent a plugin that already defines its own Invoice class from loading. This may be an acceptable behaviour change (after all, library implementers frequently add classes in order to provide new features). Second, Bill is no longer a public class; this change is more troubling since it would break existing plugins that rely explicitly on the Bill functionality.

Rather than revert the transformation, with program metamorphosis the user can *selectively* accept behavioural change by modifying the desired program model to indicate that there should be a public class called Invoice. All future program models will be compared against this new model.

Desired Visibility Model	public → {Bill, Invoice}, protected → {}
Current Visibility Model	public → {Invoice}, protected → {}
Inconsistencies	Missing public class Bill

Even after modifying the desired model, the 'Missing public class Bill' inconsistency remains; the user could choose to clear this up by, say, introducing a new version of Bill that delegates to an instance of Invoice.

Compare this user experience to that offered by traditional refactoring implementations: those typically require the user to make an all-or-nothing choice to apply an unsafe transformation after warning her that it may change behaviour. She must then determine how the program text was transformed, discern how these alterations would change program behaviour, and decide if that new behaviour is acceptable. In contrast, program metamorphosis determines how the program *behaviour* has been changed and allows the user to approve or reject those behavioural changes individually.

3.4 Reusing Equivalence Checks

One benefit of using program models rather than the predictive analyses required by preconditions is that they are often agnostic as to how the program is transformed. The name model is constructed in the same way regardless of whether a name could be captured via a rename, a field move, or a pull-up method. Similarly, it doesn't matter to the visibility model whether a class's visibility was changed because it was deleted or because it was renamed. This is advantageous because as we add new PM steps, they can reuse the same program models and get existing behaviour preservation checks "for free." While refactorings can also

sometimes exploit commonalities [7], it is much less obvious how source code for predictive analyses can be re-used.

3.5 Summary

By using postconditions and program models to check for behavioural equivalence, program metamorphosis allows users to safely compose sequences of transformational steps that may not preserve behaviour individually. This approach is more natural than that used by traditional refactorings because it does not force users to plan ahead; instead, program metamorphosis notifies them whenever they have arrived at a non-equivalent or even ill-formed program and allows them to continue transforming until the problem has been corrected. Further, this approach enables an elegant mechanism of informing users about possible behavioural changes and allows them to selectively choose which, if any, of those changes are acceptable.

4 Program Metamorphosis and Refactoring

As we have seen, our PM steps are quite different from traditional refactorings, even though they achieve similar goals. In this section, we investigate the relation between these two classes of transformations on a high level. We first take refactorings apart and show how their components relate to the components of a program metamorphosis system (Section 4.1), and then formally derive the notion of a program metamorphosis system from the resulting building blocks (Section 4.2). We establish some basic properties about this formalism (Section 4.3) and finally 'close the circle' by showing how we can build refactorings from PM steps (Section 4.4).

4.1 How Refactorings Work

Abstractly, a refactoring is a pair $\langle P, t\rangle$. P is a safety precondition that determines whether or not the refactoring is applicable to a given program. t transforms the program.

Since refactorings should preserve behaviour, P should ensure that the program has the same behaviour before and after applying t:

$$P(p) \implies [\![t(p)]\!] = [\![p]\!]$$

where $[\![-]\!]$ maps a program to its behaviour. Refactoring implementers then typically implement P such that

$$P(p) \implies V(p) \wedge V(t(p)) \wedge (p \equiv t(p))$$

i.e., the precondition $P(p)$ holds only if the input program p is well-formed ($V(p)$) and the the resulting program will both be well-formed and (in some

sense) equivalent to the input program ($p \equiv t(p)$); ideally, but not necessarily, by exhibiting precisely the same behaviour (cf. Section 2.3).

If we examine existing refactoring implementations in more detail, we observe that they implement both the validity predicates and the notion of equivalence via an intermediate step, namely the construction of a model. This model is typically a set or slice of some relevant properties of the current program; refactoring developers choose those properties so that they can check for well-formedness and predict the outcome of the transformation. For example, Griswold [5] uses a Program-Dependence Graph to determine relevant relationships that might be affected by the refactoring, while Eclipse uses a comprehensive name and type model provided by its JDT library[1]. Let us assume that we compute such a program model m with a program analysis $\mathsf{properties}$, i.e., $m = \mathsf{properties}(p)$. Then the above implication becomes

$$P(p) \implies V(m) \wedge V(t'(m)) \wedge (m \equiv t'(m))$$

(modulo overloading of our predicate V and equivalence relation $\equiv$). Here, t' is a simulation of the effect of transformation t on the program model:

$$\mathsf{properties} \circ t = t' \circ \mathsf{properties}$$

i.e., we should arrive at the same model if we first compute the program model and then apply t' as if we first transform the program and then compute a program model from the result.

In practice, refactoring implementors usually don't need to make the modified model $t'(m)$ explicit, since they can use domain knowledge to (a) re-compute only the relevant slice of the program model that might have been affected by the transformation and (b) manually deforest [16] their code to directly check for possible changes at the same time as computing the effect the transformation would have on the model.

4.2 Towards Program Metamorphosis

Such optimisations lead to tightly integrated t', V and $(\equiv)$. But if we make all three explicit, we obtain the building blocks for program metamorphosis.

To see this, recall our example from Section 3.2 of moving a method together with the field the method depends on. Let t_m be the move for the method and t_f the move for the field. Then we have that

$$V(t'_m(p)) \text{ does } NOT \text{ hold}$$

i.e, our program is ill-formed after the first transformation step (because the method can no longer see the field from its new location). However,

$$V(t'_f \circ t'_m(p)) \text{ and, moreover, } m \equiv t'_f \circ t'_m(m)$$

i.e., the composition of both transformation steps preserves behaviour. Here, we exploit that program well-formedness (V) is independent of any preceding transformations.

[1] `http://www.eclipse.org/jdt/overview.php`

4.3 Soundness and Derivation

In this section, we find that we can always construct a program metamorphosis system from an existing set of refactorings if we can decompose the refactorings appropriately, and that the resulting metamorphosis system gives the same consistency promises as the original set of refactorings. We make this more concrete in the following:

First, assume that $\mathsf{properties} : L \to \mathcal{M}$ computes a program model $m \in \mathcal{M}$ from a program $p \in L$.

Definition 1. *A* program metamorphosis system *is a tuple* $\langle \mathcal{M}, \mathsf{properties}, \equiv, V \rangle$ *such that* $V(\mathsf{properties}(p))$ *iff the program* p *is well-formed.*

To simplify our exposition, we overload $V(p) \iff V(\mathsf{properties}(p))$ and $p \equiv p' \iff \mathsf{properties}(p) \equiv \mathsf{properties}(p')$.

As we have discussed previously, our analyses and equivalence relations can be 'pessimistic' or 'optimistic'. For pessimistic metamorphosis systems we can utilise the above intuition to show a useful property regarding the strength of our consistency promises:

Definition 2. *A program metamorphosis system is* sound *wrt a language semantics* $[\![-]\!]$ *iff, for all programs* $p, p' \in L$ *such that* p' *can be reached from* p *with program metamorphosis steps,*

$$V(p) \wedge V(p') \wedge (p \equiv p') \implies [\![p]\!] = [\![p']\!]$$

Conveniently, we can construct metamorphosis systems from refactoring preconditions such that the metamorphosis systems are sound whenever the preconditions are sound. Recall our earlier decomposition of preconditions:

$$P(p) \iff V(p) \wedge V(t(p)) \wedge (p \equiv t(p))$$

If we set $(\equiv) = (\equiv_{[\![-]\!]})$, where $p \equiv_{[\![-]\!]} p' \iff [\![p]\!] = [\![p']\!]$, we have the "perfect" predicate for *any* refactoring. This relation is undecidable, so we must choose another. If we choose not to be conservative (i.e., if we do not guarantee behaviour preservation), we may pick any relation. If we are conservative, we must pick a $(\equiv) \subset (\equiv_{[\![-]\!]})$, i.e., a conservative approximation that distinguishes some programs that would be semantically equivalent. We can then immediately see the following:

Theorem 1. *Given the decomposition of refactoring preconditions* $P_1, \ldots, P_n$*, we can construct a metamorphosis system that is sound if* $P_1, \ldots, P_n$ *are conservative, and allows at least as many transformations as* $P_1, \ldots, P_n$ *allow.*

Proof. Let $(\equiv_1), \cdots, (\equiv_n)$ *be the equivalence relations used in* $P_1, \ldots, P_n$*. Then we set*

$$(\equiv) = (\equiv_1) \cup \cdots \cup (\equiv_n)$$

All $(\equiv_i)$ *are conservative approximations of* $(\equiv_{[\![-]\!]})$*, so* $(\equiv)$ *inherits this property. Furthermore, for any programs* p_1, p_2 *we have that* $p_1 \equiv_i p_2$ $(1 \leq i \leq n)$ *implies* $p_1 \equiv p_2$.

In the above, we did not specify how the program models and properties functions of the various preconditions should be combined. In theory, we can always resort to a straightforward cartesian product on the program model (as suggested by our construction of the combined $(\equiv)$), but in practice this is wasteful: most program models can be factored into common components (for example, practically all refactorings need a name model, and most need a type model). The net result of this observation is that the complexity and size of our model apparatus never increases (and often decreases) relatively to the number of transformations every time we merge two metamorphosis systems.

4.4 Back to Refactoring

Having separated refactorings into individual program transformations, equivalence predicates and program validity checks, we can now reconstruct refactorings as compositions of transformations with a post-hoc equivalence check, by slightly adjusting our combination scheme from Section 2.1:

1. Record the initial program.
2. Apply all PM steps that make up the refactoring following appropriate heuristics.
3. Determine whether the resulting program is both valid and equivalent to the initial one; otherwise roll back.

In Sections 5.2 and 6.1 we give concrete examples that illustrate this idea.

5 Program Metamorphosis in Practice

To experiment with stateful program metamorphosis, we implemented a number of prototype systems [11]. Below, we detail the most mature of our systems, a stateful program metamorphosis system for Java that functions as a plugin for the Eclipse IDE (version 3.2.2). We employ the same infrastructure that Eclipse's built-in refactorings use in order to make a comparison between the two approaches meaningful.

5.1 Program Metamorphosis in Java

We first describe our prototype's program model, our consistency promises, and the PM steps it supports, followed by a discussion of our user interface and a demonstration of the flexibility of our system compared to traditional refactorings.

Program model. Our program model includes the results of name, Use-Def, and Def-Use analyses. For name analysis, we use Eclipse's built-in bindings mechanism to determine the declaration for each use of a name and store a mapping

between names and declarations. For Def-Use and Use-Def chains, we calculate intra-procedural reaching definitions and similarly store a mapping between uses and definitions. We recompute the model when the program changes; our model equivalence test reports an inconsistency whenever the newly computed mappings do not match the original ones.

Consistency promises. Our prototype tracks whether variables, classes, type variables and methods refer to the same entities as before metamorphosis. Since it is impossible in general to determine the precise dynamic type of an expression, our system uses static types to resolve dispatch; thus, we are sometimes inaccurate when determining whether two methods refer to the same piece of functionality before and during metamorphosis. Our system may therefore conservatively issue inconsistencies where there are none; the user can review such inconsistencies and override them as (potential) behavioural change.

We further track re-ordering among read and write operations in local variables, which can arise when we move code fragments via PM-Cut and PM-Paste (see below).

PM steps. We have focussed on implementing small, composable transformations that, when combined, can match and exceed the expressive power of common refactorings. To that end, our current prototype supports the following PM steps:

- *PM-Rename*: change the name of a type or variable and its uses. This step is similar to the 'Rename' refactoring except that it uses the current program model to link names to declarations. Unlike the 'Rename' refactoring, PM-Rename allows name changes that result in name captures or other inconsistencies. Since this step does not alter the *desired* program model, we lose no information when a renaming causes names to conflict.
- *PM-Split*: take a single assignment and convert it into a declaration and initialiser (such as "x = y + 500;" $\longrightarrow$ "int x = y + 500;"). Unlike the 'Split Temporary' refactoring, this step does not introduce a new variable name for the declaration.
- *PM-Delegate*: replace a method call on implicit *this* with the same method call on another object or vice versa (e.g. "bar()" $\leftrightarrow$ "foo.bar()").
- *PM-Cut*: remove a statement, field, or method, along with its associated program model fragment, and place it in a clipboard. There is no analogue to PM-Cut in refactoring.
- *PM-Paste*: retrieve the statement, field, or method from the current clipboard and paste it and the program model fragment into a class or method body. There is no analogue to PM-Paste in refactoring.

Our PM steps act on both the AST and the program model. For example, PM-Split replaces an assignment AST node with a variable declaration node, but also updates the name mappings in the model so that each name that uses the definition now maps to the new declaration.

We have by no means implemented the complete set of useful PM steps. However, as Kiezun et al. point out [6], the fraction of 'Rename' and 'Move' refactorings among all refactorings used in practice is very high, "perhaps as high as 90% of all refactorings". We chose to provide PM-Rename and PM-Cut/PM-Paste to support these refactorings. PM-Cut and PM-Paste also permit great flexibility in program evolution and demonstrate that program metamorphosis has utility beyond mere refactoring. PM-Delegate followed from PM-Cut/PM-Paste; it is very useful for clearing up inconsistencies when moving code between different classes and methods. We implemented PM-Split to provide support for the 'Split Temporary' refactoring and to illustrate that our notion of program models scales to program properties other than name analysis mappings.

User interface. Our prototype attempts to mirror the user interface workflow of Eclipse's refactorings as much as possible. The user selects a portion of program text in the main editor and then chooses a PM step from an Eclipse menu. This brings up a modal "wizard" box that requests additional information (e.g., the new name in a PM-Rename step), if necessary. The user can then review a list of textual changes that the PM step will perform and can choose to apply or abort the step.

If the user chooses to apply the step, we ask Eclipse to perform these textual changes. We then recompute the model and compare it to the desired model, listing any differences as "Problem Markers" in the Eclipse pane for syntax errors and warnings. The user may choose to accept any of these differences as a change in program behaviour using Eclipse's "Quick Fix" interface and/or apply additional PM steps to resolve them.

In order to maintain our consistency guarantees, we must prevent the user from free-form editing the program text. While it may sometimes be possible to map arbitrary edits into appropriate program model updates (borrowing ideas from [15]), we cannot expect such approaches to work in general. Consider a program with name capture: if the user writes a new statement referencing the captured name, it is unclear which declaration she means.

5.2 Flexibility

Using the five PM steps supported by our system (cf. Section 5.1) we found that we can implement some refactorings completely, while offering partial support for others. Our prototype supports seven standard refactorings [4]: 'Rename', 'Pull Up Method', 'Pull Up Field', 'Push Down method', 'Push Down Field' (all described in Section 6), as well as 'Move Field' and 'Split Temporary'. Note that the 'Push Down' refactorings are currently limited to pushing down to a single class due to an implementation limitation (Section 6). We currently have no facility for adding or removing classes; but if the user manually adds empty classes and uninitialised fields before beginning program metamorphosis and manually deletes other classes afterward, we can support two additional refactorings, 'Tease Apart Inheritance' (Section 5.3) and 'Extract Class'. For

many other refactorings, such as 'Move Method', 'Inline Method', or 'Replace Inheritance With Delegation', our system can provide significant support.

That we do not support all standard refactorings is a limitation of our prototype and not of program metamorphosis in general. With additional PM steps (e.g. steps to introduce new classes, methods, and declarations) and a more sophisticated program model (e.g. global value numbering) we could fully support more refactorings.

Our prototype puts a similar amount of effort into preserving program behaviour as existing refactoring tools do (Section 6). Like traditional refactorings, we ensure that the final program is well-formed (largely relying on Eclipse's existing facilities to do so), preserve unique references to methods, fields, and variables, and make no attempt to maintain library APIs. Unlike Eclipse's automated refactorings, we preserve the order of reads and writes to local variables.

5.3 Teasing Apart Inheritance

Fowler [4] lists a "big refactoring" called 'Tease Apart Inheritance', for cleaning up class hierarchies that do not clearly separate responsibilities. This refactoring is hard to fully support with traditional refactoring approaches but useful for showcasing some of the strengths of our approach. For example, consider a class "NetworkServer" with subclasses "TCPChatServer" and "UDPDataServer": here we have hardwired application protocols (Chat/Data) to transport protocols (TCP/UDP).

Figure 4 illustrates this idea and the desired program evolution on an abstract level: the upper part of the figure shows the class hierarchy of "N" and its children "A×X" and "B×Y" before changing the program. Assume that the method "f" in both "A×X" and "B×Y" has the following form:

```
void f() { ...  g();  ... }
```

Since our classes "A×X" and "B×Y" combine functionality that should be handled orthogonally, we wish to tease them apart, by moving the different implementations of method "g" into a separate inheritance hierarchy. Figure 4 again illustrates this idea: We extract "g" into separate classes "X" and "Y"

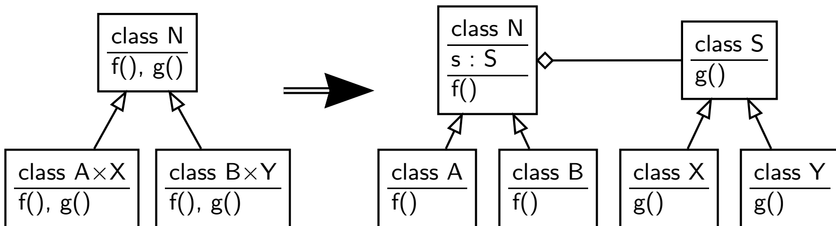

Fig. 4. Teasing Apart Inheritance: we extract functionality "X" and "Y" into a new class hierarchy underneath "S". The refactored class "N" then aggregates an instance of "S".

beneath a new, common (abstract) superclass “S” and insert a field “s” of type “S” into class “N”.

With program metamorphosis, we can do this straightforwardly: we PM-Cut and PM-Paste all relevant methods as shown in Figure 4, and then apply ‘PM-Delegate’ to the calls to “g()” in “f()”:

```
void f() { ...  s.g();  ... }
```

Our system still gives us inconsistencies for “s.g()”, since we use the static type of “s” to determine which “g” we are calling. We can address these inconsistencies easily by accepting them as a potential behavioural change.

Our current prototype provides all relevant functionality for this process, except for introducing the field “s”. Also, introducing and/or deleting classes during metamorphosis currently invalidates our consistency promises, so we must add new classes before and obsolete old classes afterwards.

We are not aware of any way to implement the above directly using only traditional refactorings, and no refactoring engine we have experimented with supports ‘Tease Apart Inheritance’ directly.

6 Prototype Correctness

As we have suggested in Sections 4 and 5.3, refactorings can always be embedded into a program metamorphosis system, and often split into smaller, more flexible parts (Section 5.2). Program metamorphosis is thus (in theory) intrinsically at least as flexible as traditional refactoring; as we have seen in Section 5.2, it is (in practice) more flexible. However, this flexibility might be a trade-off with safety: despite our argument in Section 4.3 that it is *possible* to be as safe as refactoring, it might not be *practical* to implement a program metamorphosis system that indeed achieves a comparable level of safety.

To investigate this concern, we opted to compare the safety of our PM steps with refactorings provided by an established refactoring system. Since PM steps are more fine-grained than refactorings, we constructed three standard refactorings out of the metamorphosis steps provided by our prototype. There are many ways to construct such refactorings in practice, if we include all possible automatic fixups. We chose to implement all of our refactorings in a very straightforward manner: transform, check for inconsistencies, and abort if there are any inconsistencies (simulating the effect of a refactoring precondition). While this does not exploit the inherent flexibility of program metamorphosis, it is sufficient to address our principal experimental concern, safety.

6.1 Experimental Setup

For our experiments, we paired our manually constructed refactorings with refactoring built into Eclipse 3.2.2 (since our system was developed for the Eclipse 3.2 infrastructure). Our refactorings were as follows:

- **Rename**. Our 'Rename' refactoring simply performs a PM-Rename step, but does not attempt to avoid or resolve any name capture. We configured Eclipse's Rename to rename all other relevant identifiers, including identifiers of overriding and overridden methods in super- and subclasses, which our renaming does not do implicitly. Eclipse also provides a feature that will rename occurrences of a class name in a string or external text file. This option is meant to address uses of reflection, wherein Java may instantiate a class, invoke a method, or read from or write to a variable designated by a string value. Since this mechanism is unsafe in practice, we left it disabled.
- **Pull Up Field**. Our 'Pull Up Field' refactoring moves a field to a superclass (PM-Cut followed by PM-Paste), then iterates over all subclasses of the target class to identify fields of the same name. For each such field it tests if the initialiser is identical to the initialiser of the initially selected field, and, if so, deletes the field in the subclass (per PM-Cut).
  For Eclipse's 'Pull Up Field', we instructed Eclipse to also pull up dependent methods and fields, if necessary (in practice, this should only be needed if those entities are used in the field's initialiser.)
- **Pull Up Method**. Our Pull Up Method refactoring implementation is analogous to Pull Up Field, except that we also determine all methods and fields transitively referenced in the method and pull those up afterwards.
  We configured Eclipse's Pull Up Method to also move all dependent entities.

We then instructed our system to randomly locate opportunities for applying such refactorings in a given program. Our mechanisms for choosing such opportunities were as follows:

- **Rename**: For every 'Rename', we identified a possibly renameable entity (a 'SimpleName', in Eclipse JDT nomenclature) anywhere in the program. We skipped package names because of limitations of our testing infrastructure, but included class names and names of entities external to the program (such as the 'toString()' in java.lang.Object).
  We then decided a new name as follows: with a probability of 0.5 we chose a fresh name, otherwise we chose a random name from the same compilation unit. These names were chosen the same way that renameable entities were chosen; in particular, names that occur frequently in a class had a higher probability of being chosen. If the new name was identical to the original name, we instead chose a fresh name.
- **Pull Up**: For pulling up, we identified pairs of types (interfaces, abstract classes, concrete classes) in a nontrivial supertype relationship (i.e., the classes were not identical) together with a method or field that could be moved from one type to the other, as required by the specific refactoring.

To test the correctness of a refactoring, we tested for whether the refactoring aborted, succeeded, or failed. We say that a refactoring *aborted* if the refactoring indicated that it was not applicable / would change behaviour. In traditional refactoring terms, this is usually expressed as the precondition failing. We say that a refactoring *succeeded* if the refactoring applied, and the program was both

statically well-formed and dynamically behaved the same as before, as far as we could tell (see below). We say that a refactoring *failed* if the refactoring applied (i.e., the precondition did not fail) but the resulting program was statically ill-formed or did not preserve its dynamic behaviour.

To determine whether dynamic behaviour had changed, we ran the unit test suite shipped with the programs in question. If any unit test failed, we assumed that dynamic behaviour had changed in an unintended way and that the refactoring had therefore failed.

We also automatically asserted that all non–aborted transformations had indeed modified the program and manually sampled the results to ensure that the transformations were reasonably close to our expectations.

Our specific approaches for determining refactoring results were as follows:

- **Eclipse refactoring**: For Eclipse's refactoring, we attempted to apply the refactoring (using the refactoring scripting interface) atomically. If the attempt failed (usually because a precondition failed), we marked the refactoring as *aborted*. Otherwise we ran Eclipse's own static checks on the program and any unit tests. If either the static checks or the unit tests failed, the refactoring *failed*, otherwise it *succeeded*.
- **Program metamorphosis**: For our own refactorings, we applied all relevant transformations (usually several) in sequence, disregarding any inconsistencies until the end. After we had finished transforming, we ran our own inconsistency checks as well as Eclipse's static checks. If either indicated an error or inconsistency, we *aborted*. Otherwise we ran the unit tests to determine whether the refactoring had *failed* or *succeeded*.

Note that we interpreted the results of Eclipse's own correctness checks differently for program metamorphosis and traditional Refactoring. This reflects the program metamorphosis philosophy and highlights an advantage of our approach: by definition, a traditional refactoring must preserve behaviour if its preconditions trigger – in particular, it must produce a well-formed program. Program metamorphosis, on the other hand, need only be able to determine whether the program is well-formed or not *after the fact*. As we observed with Eclipse, this allows us to exploit traditional IDE correctness checks to augment our own checks for program model equivalence (Section 5.1).

6.2 Results

We ran our experiments against the following programs:

- Functional Analyzer [10], a flexible tool for fast analysis of trace information and similar numerical data, developed by one of the authors (7714 loc[2]).
- Apache Commons: Discovery 0.4, a library for detecting and managing plugins, developed by the free Apache Commons project (2543 loc).

[2] Lines of non-comment non-whitespace source code, computed with `sloccount`.

- Apache Commons: Validator 1.3.1, a general-purpose validation library for structured data, particularly XML (8874 loc).
- Apache Commons: Chain 0.4, a chain-of-responsibility implementation, again part of the Apache Commons project (8010 loc).
- Apache Commons: Digester 1.8, a configurable XML configuration file interpreter (12342 loc).

We chose the above programs by availability and presence of substantial unit test suites.

For each experiment, we configured our system to perform 200 random transformations for each refactoring. Table 5 summarises our results.

As we can see from our results, our (fairly straightforwardly) PM-scripted refactorings are competitive with Eclipse's. In the majority of the cases we tested, both systems behaved equivalently. Where they didn't, the differences were mostly due to Eclipse being more flexible by providing additional fixups or Eclipse being less conservative (particularly when pulling up) and thus being simultaneously more flexible and more error-prone. For 'Pull Up', our primitive dependency analysis was sometimes fooled, most commonly by `this` references, resulting in additional aborts. In all instances that that we observed, a human programmer, driven by our inconsistencies, would have been able to identify and rectify the situation straightforwardly. In other instances, less-than-ideal interfacing between our module and the Eclipse parser prevented our prototype from matching up code from before and after a transformation (particularly in Rename). With respect to the focus of our tests, we observed that PM-scripted refactorings were safer than Eclipse's traditional refactorings: averaging over all of our tests, the cases in which the PM-scripted refactorings failed and Eclipse's refactorings succeeded or aborted made up 0.1%, while the cases in which Eclipse's refactorings failed and the PM-scripted refactorings aborted or succeeded made up 24.4% of all tests.

- **Pull Up Field.** Pulling up, Eclipse attempts to merge fields from all subclasses, whether or not those fields have the same initialisers. This frequently introduces bugs, not all of which are caught by unit tests. If the common fields' types mismatch, Eclipse aborts, while our simple pull-up heuristic skips the fields if their initialisers differ, accounting for a few cases in which our PM-scripted refactoring is more flexible. In other cases, Eclipse implicitly changed field visibility (from private to protected) if needed, which was not part of our PM scripting.
- **Pull Up Method.** For 'Pull Up Method', both refactoring implementations failed consistently when pulling up unit test methods into superclasses for which not all subclasses satisfied the test, in the Commons Validator. When pulling up methods, Eclipse again suffered from its implicit merging of fields when pulling up dependent entities, while our PM-scripted refactoring's refusal to implicitly change visibility accounted for much of its lack of flexibility. Eclipse's 'Pull Up Method' further changes requests to pull up a method into an interface into a request to add a method

	Identical				More Flexible		More Failures	
Refactoring	**abort**	**success**	**failure**	**total**	**PM**	**Eclipse**	**PM**	**Eclipse**
Pull Up Field								
Functional Analyzer	9.0%	4.0%	0.0%	13.0%	2.0%	7.0%	0.0%	80.0%
Commons Discovery	13.5%	9.0%	0.0%	22.5%	0.0%	37.0%	0.0%	40.5%
Commons Validator	31.5%	4.5%	0.0%	36.0%	18.5%	32.5%	0.0%	31.5%
Commons Chain	16.5%	0.0%	0.0%	16.5%	1.5%	40.0%	0.0%	43.5%
Commons Digester	3.0%	19.0%	0.0%	22.0%	0.0%	43.0%	0.0%	35.0%
average	14.7%	7.3%	0.0%	22.0%	4.4%	31.9%	0.0%	46.1%
Pull Up Method								
Functional Analyzer	55.0%	3.0%	0.0%	58.0%	0.0%	20.5%	0.0%	21.5%
Commons Discovery	51.5%	0.0%	0.0%	51.5%	0.0%	20.0%	0.0%	28.5%
Commons Validator	46.0%	29.0%	7.5%	82.5%	0.0%	9.0%	0.0%	8.5%
Commons Chain	51.5%	0.5%	0.0%	52.0%	1.5%	5.0%	0.0%	42.5%
Commons Chain	46.0%	4.5%	2.5%	53.0%	0.0%	19.5%	0.0%	27.5%
average	50.0%	7.4%	2.0%	59.4%	0.3%	14.8%	0.0%	25.7%
Rename								
Functional Analyzer	17.0%	60.5%	0.5%	78.0%	0.0%	21.5%	0.0%	0.5%
Commons Discovery	29.0%	59.5%	2.0%	90.5%	0.0%	9.0%	0.0%	0.5%
Commons Validator	30.5%	60.5%	1.5%	92.5%	0.5%	5.5%	0.5%	1.5%
Commons Chain	43.0%	50.0%	1.0%	94.0%	0.5%	2.0%	0.5%	3.0%
Commons Chain	29.0%	59.0%	1.0%	89.0%	0.5%	9.5%	0.5%	1.0%
average	29.7%	57.9%	1.2%	88.8%	0.3%	9.5%	0.3%	1.3%

Fig. 5. Benchmarking results for Eclipse's refactoring suite (Eclipse) and refactoring scripted from program metamorphosis steps (PM). **Identical** identifies cases in which both tools behaved equivalently. **More Flexible** identifies cases in which one tool permitted a transformation while the other tool aborted that transformation. **More Failures** identifies cases in which one tool caused behavioural change. Note that **Identical(total)**, **More Flexible** and **More Failures** sometimes add up to more than 100% in cases where both tools performed the transformation but one tool produced an incorrect result (we counted this as the correct tool being both safer and more flexible). Considering cases where Eclipse failed, this accounts for all of the cases in which PM was more flexible in 'Pull Up Field', as well as for 1% of the 'Pull Up Method' cases in the Commons Chain. Considering cases where PM-scripted refactoring failed, this accounted for two cases in 'Rename' (cf. our discussion).

declaration of the same interface to the interface, again adding to its flexibility (an actual 'Pull Up Method' into an interface is only possible for abstract methods).

- **Rename.** For renaming, our system primarily suffered from two limitations: first, our prototype will not refactor constructors in some cases, and secondly, we do not enforce the override status of overriding methods in subclasses.

  Refactoring constructors requires renaming the class and all related constructors. A limitation of the Eclipse parser that we have not yet addressed

sometimes prohibits this in classes with multiple constructors; this issue accounts for 3% of the aborted rename attempts (total) in the Functional Analyzer, 10.5% in Commons Discovery, 13% in the Commons Validator, 16.5% in the Commons Chain package, and 12% in the Commons Digester. Note that these failures also account for some of its increased safety in the presence of reflection. Since reflection allows classes to be looked up by names read from external files, it is notoriously hard to support in any kind of refactoring process. (All of our test cases utilise reflection to some extent.)

Another current limitation is that our inconsistency checks do not enforce the overriding status of methods when renaming methods of a subclass. In the presence of an `@Override` annotation, this usually leads to static errors, but in two cases it allowed the PM-scripted refactoring to introduce a dynamic failure. We expect to extend our program model to add either explicit 'method-X-overrides-method-Y' information or global value numbering to increase the strength of our correctness promises overall.

We also experimented with 'Push Down Method' and 'Push Down Field'. Due to an unresolved issue in our prototype, our 'Push Down' operations are currently overly conservative when pushing to multiple subclasses: copying (rather than cutting and pasting) generates 'fresh' methods and fields, resulting in spurious inconsistency warnings that cannot be accepted as behavioural change. Conversely, Eclipse's 'Push Down' refactoring cannot be constrained to push down to one particular subclass: instead, it always pushes down to all immediate subclasses, though users can interactively choose to suppress parts of the textual diff after the refactoring has terminated. We could thus not directly compare the two sets of functionality, though we have no reason to assume that a corrected PM-scripted 'Push Down' would ultimately exhibit correctness or performance characteristics different from the PM-scripted 'Pull Up'.

While our results overall indicate that our scripted refactorings are less flexible than Eclipse's refactorings, we note the following:

- Our prototype is, on average, safer than Eclipse's refactorings.
- Our prototype permits us to quickly script refactorings that are as flexible as Eclipse's refactorings in most of the cases we examined, without including any automated fixups or complex analyses as part of the scripting.

6.3 Practicality

One goal of our Java prototype is to examine whether program metamorphosis is practical to implement and useful for evolving real-world programs. Here, we evaluate our prototype in terms of code size and resource consumption.

The main component of the memory cost for program metamorphosis is the need to keep an AST of the entire program in memory at all times. Our prototype requires two copies of the full AST in memory during equivalence

checking. Using the the Eclipse JDT's built-in memory queries, we have determined that a single instance of the Functional Analyzer AST requires approximately 4MB of memory, which we consider to be acceptable on modern machines.

Our prototype (excluding unit tests) consists of 3829 lines of Java across 53 files and relies significantly on Eclipse's infrastructure to perform program analysis and to interact with the user. This shows that a useful set of PM steps can be implemented in a relatively small amount of code and that PM can be compatible with existing program analysis frameworks and program evolution tools. For this reason, we have favoured ease of implementation and tight integration with Eclipse over speed. For example, we used the JDT's built-in name analysis even though it requires re-parsing to get updated analysis. We could reduce our runtime overhead by comparing only altered parts of the program and recomputing program models lazily. This would decrease execution times and memory usage at the cost of added complexity.

To ensure that our prototype is practical for interactive use, we measured execution times for the correctness tests from Section 6. We summarise these results in Figure 6. All experiments were run on a 2.4GHz Intel Core 2 Quad with 4GB of RAM, running Java 1.6.0_03-b05 on Ubuntu 7.1 with Linux 2.6.24.

		Eclipse			PM		
Refactoring	**Program**	**min**	**avg**	**max**	**min**	**avg**	**max**
Pull Up Field	Functional Analyzer	0.11s	0.32s	1.15s	2.05s	2.29s	2.81s
	Commons Discovery	0.13s	0.24s	0.52s	0.86s	0.97s	1.37s
	Commons Validator	0.18s	0.45s	0.96s	2.70s	3.07s	4.55s
	Commons Chains	0.36s	0.47s	0.83s	2.39s	2.49s	2.64s
	Commons Digester	0.15s	0.31s	1.51s	3.09s	3.44s	7.87s
	total	0.11s	0.36s	1.51s	0.86s	2.45s	7.87s
Pull Up Method	Functional Analyzer	0.12s	0.32s	0.83s	1.90s	2.55s	3.35s
	Commons Discovery	0.14s	0.32s	2.95s	**n/a**	**n/a**	**n/a**
	Commons Validator	0.22s	0.47s	1.93s	2.73s	3.79s	11.65s
	Commons Chains	0.39s	0.64s	1.74s	2.18s	2.38s	2.59s
	Commons Digester	0.19s	0.55s	1.62s	3.09s	4.83s	10.43s
	total	0.12s	0.46s	2.95s	1.90s	3.39s	11.65s
Rename	Functional Analyzer	0.09s	0.34s	2.71s	1.10s	1.36s	2.92s
	Commons Discovery	0.02s	0.15s	0.85s	0.43s	0.53s	1.53s
	Commons Validator	0.06s	0.20s	0.69s	1.46s	1.71s	1.98s
	Commons Chains	0.06s	0.26s	0.98s	1.14s	1.35s	2.12s
	Commons Digester	0.12s	0.28s	1.66s	1.72s	1.95s	2.58s
	total	0.02s	0.25s	2.71s	0.43s	1.38s	2.92s

Fig. 6. Minimum, average, and maximum refactoring execution times, for both Eclipse's built-in refactorings and program metamorphosis

For program metamorphosis, the execution time is the sum of the execution times of all intermediate steps. The **total** line gives the overall minimum, maximum, and average of the averages (as summarised per program).

Our unoptimised prototype executes most transformation steps well within the time limits of what we can expect from an interactive tool. While some transformations may take more than two seconds to complete overall, note that both of our pull-up refactorings are multi-step transformations in program metamorphosis, as we explained in Section 6; except for two Renames (one in the Functional Analyzer and one in the Commons Digester), no individual transformation execution time was more than 2.5s per PM step (including re-parsing and AST re-matching for the entire program after each step).

7 Related Work

There is a large body of related work on refactoring (cf. [9] for a survey), including many implementations, such as HaRe [8] and Eclipse [14]. The observation that more information than immediately visible to the eye is needed to perform correct transformations was already employed by Griswold [5], who used Program Dependence Graphs [3] for this purpose. These systems consider refactorings to be individual macroscopic transformations. Some other program transformation approaches [1, 17] look specifically for atomic transformations, but remain entirely semantics-preserving.

Composing transformations to achieve a certain goal is the central theme of AI Planning (cf. [12] for a high-level overview). The composition of refactorings in particular has also been considered [7], but only for traditional approaches to refactoring, without allowing intermediate invalidation of correctness properties.

8 Conclusion

We have presented a novel approach to program evolution in which users interactively combine small program transformations, *PM steps*, while a consistency checking mechanism tracks behavioural change that they introduce. As part of this process, users can choose to explicitly alter behaviour rather than to preserve it. Since our approach differs from refactoring (a) by allowing users to transform more liberally and (b) by permitting explicit behavioural change, we give it a different name, *program metamorphosis*. We have further described an Eclipse plugin that implements program metamorphosis for Java. Our experimental results suggest that program metamorphosis is a practical and viable approach for supplanting traditional machine support for refactoring.

Acknowledgements

The authors are indebted to Daniel von Dincklage, William Griswold, Jeremy Siek, Philipp Wetzler, and the anonymous ECOOP, POPL and ICFP referees for their valuable feedback on this work.

References

[1] Arsac, J.J.: Syntactic source to source transforms and program manipulation. Commun. ACM 22(1), 43–54 (1979)
[2] Beck, K.: eXtreme Programming eXplained, Embrace Change. Addison-Wesley, Reading (2000)
[3] Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
[4] Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading (1999)
[5] Griswold, W.G., Notkin, D.: Program Restructuring as an Aid to Software Maintenance. Technical report, Univ. of Wash (1990)
[6] Kiezun, A., Fuhrer, R.M., Keller, M.: Advanced Refactoring in Eclipse: Past, Present and Future. In: First Workshop on Refactoring Tools, Berlin (2007), `https://netfiles.uiuc.edu/dig/RefactoringWorkshop/Presentations/AdvancedRefactoringInEclipse.pdf`
[7] Kniesel, G., Koch, H.: Static composition of refactorings. Sci. Comput. Program. 52(1-3), 9–51 (2004)
[8] Li, H., Reinke, C., Thompson, S.: Tool Support for Refactoring Functional Programs. In: Jeuring, J. (ed.) ACM Sigplan Haskell Workshop, pp. 27–38 (2003)
[9] Mens, T., Tourwe, T.: A survey of software refactoring. IEEE Trans. Softw. Eng. 30(2), 126–139 (2004)
[10] Mytkowicz, T., Sweeney, P.F., Hauswirth, M., Diwan, A.: Time interpolation: So many metrics, so few registers. In: MICRO 2007: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, Washington, DC, USA, pp. 286–300. IEEE Computer Society Press, Los Alamitos (2007)
[11] Reichenbach, C., Diwan, A.: Program Metamorphosis. Technical Report CU-CS-1036-07, University of Colorado at Boulder (2007)
[12] Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 2nd edn. Prentice-Hall, Englewood Cliffs (2003)
[13] Schäfer, M., Ekman, T., de Moor, O.: Sound and Extensible Renaming for Java. In: Kiczales, G. (ed.) 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008). ACM Press, New York (2008)
[14] Shavor, S., D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., McCarthy, P.: The Java Developers Guide to Eclipse. Addison-Wesley, Reading (2003)
[15] Taneja, K., Dig, D., Xie, T.: Automated detection of API refactorings in libraries. In: ASE 2007: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 377–380. ACM Press, New York (2007)
[16] Wadler, P.: Deforestation: Transforming programs to eliminate trees. Theoretical Computer Science 73, 344–358 (1990)
[17] Ward, M.P., Zedan, H.: MetaWSL and Meta-Transformations in the FermaT Transformation System. In: COMPSAC (1), pp. 233–238 (2005)