ABSTRACT

Adaptive programming (AP) is a programming paradigm for expressing structure-shy computations over semi-structured data graphs. Structure-shyness means that adaptive programs hard code a minimal set of assumptions about the structure of their input. Because of this, adaptive programs are more susceptible to unsafe evolutions; evolutions that jeopardize the correctness of adaptive programs yet go uncaught. In this paper we study the evolution of adaptive programs and present two complementary approaches for controlling their unsafe evolution: a language for expressing application-specific constraints on the runtime behavior of adaptive programs, and a stricter notion of compatibility between the parts of an adaptive program, that does not sacrifice the expressiveness of the AP paradigm.

Categories and Subject Descriptors

D.2.4 [SOFTWARE ENGINEERING]: Software/Program Verification—Assertion checkers, Programming by contract, Reliability; D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features; D.1.2 [PROGRAMMING TECHNIQUES]: Automatic Programming—Program modification, Program verification; F.3.1 [LOGICS AND MEANINGS OF PROGRAMS]: Specifying and Verifying—Reasoning about Programs—Assertions, Specification techniques

General Terms

Languages, Design, Reliability

Keywords

Adaptive Programming, Evolution, Assertion

1. INTRODUCTION

The key observation behind the development of the Adaptive Programming paradigm (AP) [14] is that methods tend to hard code assumptions about the structure of their input that are not inherently relevant to their “function”. For example, the method area shown in Listing 1 hard codes the assumption that the radius is wrapped in a dimensions object which is a part of a Circle object.

The hard coded assumptions tighten the coupling between a method and the structure of its input, leading to less generic methods. This hinders the maintainability of programs, because evolving the structure of the input requires changes to methods. Furthermore, the reduced genericity forces developers to reuse methods via copy and paste. Hence, complicating the process of evolving methods and making it rather error prone.

One approach for restoring genericity, is to decouple the method from the structure via the use of a behavioral interface to the structure [1]. For example, the method area shown in Listing 2 does not make any assumptions about the structure of its input, except for implementing the interface IHasRadius. The downside of this approach, besides verbosity, is that the structural assumption is still hard coded, but in another method.

The approach taken by AP for writing generic methods (called adaptive methods or adaptive programs) is to hard code the minimal necessary structural assumptions about the input. For example, the method area shown in Listing 3 works for any T provided that it can reach a field called radius that can reach a double.

1 Following the law of Demeter
double area(T o) 
{ AreaVisitor v = new AreaVisitor();
    ExecuteAdaptiveProgram(
        "from * via [radius] to double", o, v);
    return v.area; }

class AreaVisitor extends Visitor 
{ double area;
    before(double host) 
    { area = host * PI; }
}

Listing 3: Structure Shy Area Function

An adaptive method comprises three loosely coupled parts: a traversal strategy, an input object, and a behavior. The traversal strategy is a regular-expression-like specification that selects a set of paths from the schema of the input object. The traversal strategy should be the only place where structural assumptions about the input schema are hard coded. In this way, the traversal strategy plays the role of an interface between the adaptive method and the input schema. The behavior is a set of collaborating type indexed methods\(^2\). In this paper, we consider an abstract incarnation of AP where input objects are instances of semi-structured data graphs. An adaptive method is executed via a depth first traversal of its input object. Along this traversal, the appropriate methods in the behavior are invoked. An adaptive method is considered legal if its strategy is compatible with the input schema. That is, the input schema meets all of the structural assumptions made by the strategy. A strategy and an input schema are incompatible iff the set of paths selected by the strategy from the input schema is empty. The selection of the “correct” set of paths is an important factor in the overall correctness of the adaptive program. In this paper, we choose to focus on the correctness of the selected paths, discarding other sources of incorrectness of adaptive programs. A strategy that selects the correct set of paths is called a correct strategy.

It is worth mentioning that the strategy does not have to fully specify every single detail of the traversal. This is the reason why adaptive methods seamlessly adapt to evolutions of their input schema\(^3\). Also, the runtime overhead of the traversal is often optimized by specializing the adaptive method for a specific input schema\(^4\). Finally, due to their organization as traversals, adaptive programs are also retargetable to different execution platforms\(^5\) (e.g. single and multi-core architectures).

1.1 Evolution of Adaptive Programs

Evolution of an adaptive program may include not only changes to its input schema but also to its strategy as well as its behavior. Figure 1(a) is essentially a Venn diagram that illustrates the possible effects of evolution on adaptive programs. The universe is all syntactically well formed adaptive programs. The continuous circle contains all legal adaptive programs. The dashed circle contains all correct adaptive programs for some application specific correctness criteria that the adaptive programming system might be unaware of. For example, running the adaptive area method shown in Listing 3 with an input object that contains two fields with the name radius can lead to an incorrect result despite the fact that the input object satisfies the structural constraints imposed by the adaptive method.

The arrows represent various kinds of evolutions of adaptive programs. Starting with a correct an legal adaptive program, evolution can lead to either:

1. Another correct adaptive program. We call this kind of evolution safe.
2. A legal but no longer correct adaptive program. We call this kind of evolution unsafe or dangerous.
3. An illegal adaptive program. We call this kind of evolution illegal.

![Figure 1: Evolution of Adaptive Programs](image)

1.2 Problem

The problem that we tackle in this paper is the unsafe evolution of adaptive programs. Further analysis reveals that it is caused by the following two problems: first, the correctness criteria of an adaptive program is unknown to the adaptive programming system. Judging the (in)correctness of a certain adaptive program requires awareness of its specific correctness criteria. Developers who are not fully aware of their application’s correctness criteria (e.g. new developers or developers in a big team) or developers not paying enough attention to a complex enough correctness criteria might unsafely evolve their adaptive programs. Since the adaptive programming system is also unaware of the application’s specific correctness criteria, it cannot detect unsafe evolutions as well, and they can go uncaught. This is a problem of virtually every programming paradigm, here instantiated in the context of the adaptive programming paradigm.

The second problem is that adaptive programming adopts a very permissive approach for judging the “compatibility” of its three parts, this can hide incorrectness. For example, consider an adaptive program with a single method indexed with a type that is not mentioned in its strategy at all. This means that given an input schema that is identical to the strategy, the strategy will select a non empty set of paths, discarding other sources of incorrectness.

1.3 Contributions

This paper aims at controlling the unsafe evolutions of adaptive programs. To that end, we propose solutions for the two aforementioned problems. As for the first problem, developers should be able to declare a correctness criteria for

\(^2\)The behavior can be also viewed as a set of collaborating aspects where the type index of the method represents the pointcut and the method body representing the advice.

\(^3\)In this paper, we consider an abstract incarnation of AP where input objects are instances of semi-structured data graphs.

\(^4\)Finally, due to their organization as traversals, adaptive programs are also retargetable to different execution platforms.

\(^5\)The adaptive area method shown in Listing 3 with an input object that contains two fields with the name radius can lead to an incorrect result despite the fact that the input object satisfies the structural constraints imposed by the adaptive method.
their specific application. The declared correctness criteria can only be based on the context(s) in which certain advice executes and how often it does so. The inner shaded region shown in Figure 1(b) contains those adaptive programs that violate their correctness criteria and hence deemed illegal.

It is worth mentioning that declaring the correctness criteria is not enough for closing the gap between correct and legal programs; besides the fact that developers can declare the wrong correctness criteria, the declared correctness criteria remains only an approximation for the real correctness criteria. The reason being that the real correctness criteria might be based on the meaning of the entire computation in another world. For example, the real “correctness criteria” might be based on the wall-clock time between the execution of two advices which is a poor fit for adaptive programming simply because there is no notion of wall-clock time, per say, in the adaptive programming model.

As for the second problem, we propose a stricter notion of compatibility that encompasses all three parts of an adaptive program and excludes those adaptive programs containing some form of a “conceptual mismatch”, regardless of their correctness. The tricky part here is preserving the expressiveness of the adaptive programming paradigm. Figure 1(c) illustrates the effect that a stricter notion of compatibility might have on controlling unsafe evolutions of adaptive programs by shrinking the original continuous circle containing legal adaptive programs to the inner one. The shaded region between the inner and outer continuous circle contain those adaptive programs that became illegal as a result of incorporating a stricter notion of compatibility.

1.4 Organization

The rest of this paper is organized as follows: Section 2 gives a background of AP through a running example that we shall be using. In Section 3 we present a comprehensive study of the evolution of adaptive programs with the goal of identifying unsafe evolutions. In Section 4 we present a language for declaring approximate correctness criteria. In Section 5 we present our stricter notion of compatibility. In Section 6 we examine the effect of our proposed solutions on the evolution of adaptive programs. In Section 7 we present some of the related work. Section 8 concludes the paper.

2. ADAPTIVE PROGRAMMING

As a concrete example of an adaptive program, consider the task of modeling a bus route. A bus route has one bus, to start with. The bus has a driver and passengers. Our task is to find the number of people, including the driver, on board the bus.

Figure 2(a) shows the schema for the input data graph shown in Figure 2(d). Figure 2(b) shows a strategy which says: go from a BusRoute object to a Person object via either a Driver object or a Passengers object. Listing 3 shows the behavior, which contains two methods (or advices): one is attached to BusRoute, which initializes a global counter, and the other is attached to Person, and it increments the counter by 1.

Before we can execute our adaptive program, we must turn the underspecified traversal specification (the strategy) into a fully specified traversal specification that we call the traversing graph; Figure 2(c) shows the traversal graph for our example. It is computed by replacing every edge, from a node S to a node T, in the strategy graph, with the maximal subgraph of the input schema that contains all nodes reachable from S and can reach T. For example, the strategy edge from BusRoute to Driver is replaced with the subgraph that contains BusRoute, Bus, and Driver. The traversal graph nodes drawn with a thick line correspond to the nodes from the strategy.

To execute our adaptive program on the data graph shown in Figure 2(d) (which satisfies the input schema) we also need both the traversal graph and the behavior, as indicated in Listing 3. The execution proceeds as follows: starting with the object busRouteObj in the input data graph as the current object and the singleton set containing the source node (whose label is BusRoute) which is the type of the current object) in the traversal graph as the current set of traversal graph nodes; we first execute any advices attached to BusRoute in the behavior. In our example, there is one such advice attached to BusRoute. Then, we go through the children of the current object (busRouteObj in our example). For each child object we identify the set of traversal graph nodes that are both a child of any node in the current set of traversal graph nodes and whose label is the same as the type of current child object. In our example, the first (and only) child to consider is busObj and the set of traversal graph nodes that satisfy both conditions contains the two nodes in the traversal graph labeled Bus. In case the set of

![Figure 2: Bus Route Example](image)

```java
class CountPeopleOnBoard
{
    private int noOfPeopleOnBoard;
    before(BusRoute busRoute)
    {
        noOfPeopleOnBoard = 0;
    }
    before(Person person)
    {
        noOfPeopleOnBoard += 1;
    }
}
```

Listing 4: Counting People on Board
traversal graph nodes is empty we proceed to the next child object. When all children are considered, we simply return. In case the set of traversal graph nodes is not empty, we recursively execute the above procedure. Listing 5 shows the pseudocode for executing an adaptive program.

To see how this program can adapt to a change in the input schema, consider adding a coordinator for the BusRoute. Figure 3 shows the evolved schema. We observe that the traversal graph for our adaptive program remains the same. Therefore, the runtime behavior of our program remains unchanged.

Listing 5: Adaptive Program Execution

```java
def ExecuteAdaptiveProgram(tgNodes, obj, beh):
    type = obj.getType()
    advice = beh.getAdviceForType(type)
    if (advice != Empty):
        advice.fire(obj)
        foreach (child : obj.getChildren())
            nextTgNodes = GetNextTgNodes(tgNodes, child, getType())
            if (nextTgNodes != EmptyList):
                ExecuteAdaptiveProgram(nextTgNodes, child, beh)
    return
```

GetNextTgNodes(currentTgNodes, label)

```java
    nextTgNodes = EmptyList
    foreach (tgNode : currentTgNodes)
        nextTgNodes.append(tgNode.getChildren(label))
    return
```

Figure 3: Bus Route Schema with a Coordinator

Another change to the input schema that our adaptive program can seamlessly handle is to add an Operator node between Bus and Driver. Figure 4 shows the evolved schema. Although the traversal graph changes in response to this evolution, the change does not result in any change to the runtime behavior of our adaptive program. The reason is that Operator is not advised.

Figure 4: Bus Route Schema with a Coordinator and an Operator

3. EVOLUTION OF ADAPTIVE PROGRAMS

In this section, we study the effect of evolution on the runtime behavior of adaptive programs. Listing 5 shows that the runtime behavior of an adaptive program depends, not only on its traversal graph (hence on the input schema and the strategy graph) and its behavior, but also on a specific input data graph. This brings up an interesting question: how do we represent the runtime behavior of an adaptive program for all possible input data graphs? Having answered this question, we delve into a study of the different classes of impact that evolution might have on the runtime behavior of a correct and legal adaptive program.

3.1 Runtime Behavior Representation

Two key observations can be made from Listing 5 about the runtime behavior of an adaptive program. The first observation is that all `tgNodes` are labeled with the type of the `obj`. This invariant is preserved in the recursive call and must be satisfied at the first invocation of `ExecuteAdaptiveProgram`. Therefore, an advice can only be executed if the type it is attached to shows up in the traversal graph. Furthermore, an advice attached to type `T` can be executed in the context of another advice attached to type `S` only if the traversal graph has nodes labeled `S` and `T` and `T` is reachable from `S`. The second observation is that line 5 where advices are executed is the only line whose execution can be externally observed. Therefore, only invocations of `ExecuteAdaptiveProgram` at objects with advised types can be externally observed.

The first observation tells us that the traversal graph contains all the necessary information for representing the runtime behavior of its adaptive program. The second observation tells us that the traversal graph contains some extra information that is irrelevant to the observable runtime behavior of its adaptive program. Based on this information we conclude that smoothing out non-advised nodes from the traversal graph yields us the most appropriate representation of the runtime behavior of an adaptive program. We call this representation the smoothed traversal graph.

The smoothed traversal graph is a multi-graph that contains only the advised nodes from the traversal graph. For every distinct, direct path connecting an advised traversal graph node `S` to another advised traversal graph node `T`, an edge is added from `S` to `T` in the smoothed traversal graph. A path in the traversal graph is represented by its set of edges rather than its sequence of nodes. A path is called direct if it contains exactly two advised nodes: one at its source and one at its target. A direct path connecting an advised traversal graph node `S` to another advised traversal graph node `T` represents a situation in which the advice attached to `T` executes right after the advice attached to `S`.

Figure 5 shows the smoothed traversal graph for the the program...
bus route example mentioned in the. It contains the two advised nodes: BusRoute and Person. All other nodes are smoothed out. It also contains three edges connecting the two nodes representing the three different situations that the advice attached to person can execute right after the advice attached to BusRoute. The three paths representing these situations are:

- { (BusRoute, Bus), (Bus, Driver), (Driver, Person)}.  
- { (BusRoute, Bus), (Bus, Passengers), (Passengers, PersonList), (PersonList, Person)}.  
- { (BusRoute, Bus), (Bus, Passengers), (Passengers, PersonList), (PersonList, PersonList), (PersonList, Person)}.

\[ \text{BusRoute} \rightarrow \text{Person} \]

(a) BusRoute (b) Person

Figure 6: Advice Execution Contexts

The contexts in which an advice is executed are also evident from the smoothed traversal graph. The contexts in which an advice attached to type T executes are fully represented by the maximal subgraph of the smoothed traversal graph containing only those nodes that can reach a node labeled T. For example, the contexts in which the advice attached to BusRoute executes, contain a single node labeled BusRoute and is shown in Figure 6(a). The contexts in which an advice attached to Person executes contain the two nodes BusRoute and Person and is shown in Figure 6(b).

### 3.2 Impacts of Evolution on Adaptive programs

The impact of evolution on the smoothed traversal graph falls into one of the following three categories:

#### 3.2.1 No Impact

Our first evolution example of adding a Coordinator to the BusRoute falls into this category. As we mentioned before, the traversal graph remains unchanged. Hence the smoothed traversal graph shown in Figure 5 remains unchanged too.

Our second evolution example of adding an Operator node between Bus and Driver falls into this category as well. Because even though an Operator node is inserted between Bus and Driver in the traversal graph, the newly inserted node gets smoothed out because it is not advised, we end up with the same smoothed traversal graph as before evolution.

In general, evolving the input schema by adding non-advised nodes, smoothing out a non-advised node, or reordering two non-advised nodes does not impact the runtime behavior of the adaptive program either because the changes do not make their way to the traversal graph (as in the first example), or they get smoothed out (as in the second example). Evolutions leading to this kind of impact are considered safe because they do not change the runtime behavior of the program, on which correctness is based.

#### 3.2.2 Minor Impact

Suppose that we were to allow many buses on the same route. The evolved input schema is shown in Figure 7(a) an unadvised node BusList is inserted, which has a self loop. Technically, this allows an infinite number of ways to reach a Bus object from a BusRoute object; by going through one, two, three, or any other number of BusList objects, whereas previously there was only one way to do so.

\[ \text{BusRoute} \rightarrow \text{BusList} \rightarrow \text{Bus} \]

Driver

(a) Bus Route with Many Buses

\[ \text{Person} \rightarrow \text{PersonList} \]

(b) Smoothed Traversal Graph

Figure 7: Bus Route with Many Buses

On the other hand, we observe that a Person object remains accessible only in the context of either a Driver object or a Passengers object and that both Driver and Passengers objects are only accessible in the context of a BusRoute object. This is essential for the advice executing at Person (Listing 1) because it relies on the fact that the noOfPeopleOnBoard counter is initialized by the advice executing at BusRoute.

In other words, even though the contexts in which both advice execute remain unchanged, the advice attached to Person is executed in more situations than before. This is evident in the evolved smoothed traversal graph shown in Figure 7(b) there are six incoming edges to Person compared to only three incoming edges before evolution. The six edges correspond to the following traversal graph paths:

- { (BusRoute, BusList), (BusList, Bus), (Bus, Driver), (Driver, Person) }
- { (BusRoute, BusList), (BusList, BusList), (BusList, Bus), (Bus, Driver), (Driver, Person) }
- { (BusRoute, BusList), (BusList, Bus), (Bus, Passengers), (Passengers, PersonList), (PersonList, Person) }
- { (BusRoute, BusList), (BusList, BusList), (BusList, Bus), (Bus, Passengers), (Passengers, PersonList), (PersonList, Person) }
- { (BusRoute, BusList), (BusList, BusList), (BusList, Bus), (Bus, Passengers), (Passengers, PersonList), (PersonList, Person) }

Another example of a minor impact is to add a Simulator (to train drivers) with a Driver and an Instructor to the BusRoute. Again, this evolution does not change any
of the contexts in which an advice executes. However, the situations in which the advice attached to Person increase as the number of ways of going from a BusRoute to a Driver increase.

In general, an evolution that results in a change to the number of ways that a node can reach another node in the smoothed traversal graph without changing the advice execution context constitutes a minor impact to the smoothed traversal graph.

An evolution leading to a minor impact is not always safe. For example, one can argue that our first example of minor impacts is safe because we will be counting the number of people on board of all buses in the route and that is the correct thing to do. On the other hand, the second example, we will also be counting the drivers who are only training and not on board of any real bus and hence can be deemed unsafe. Therefore, evolutions leading to minor impacts to the smoothed traversal graph need to be controlled.

### 3.2.3 Drastic Impact

Simply speaking, a drastic impact involves a change to at least one advice execution context. Similar to evolutions leading to minor impacts, those drastic impacts can be either safe, unsafe, or even illegal (we shall discuss illegal evolutions in Section 4). Therefore, evolutions leading to drastic impacts need to be controlled.

As an example of a safe evolution leading to a drastic impact, consider adding a coordinator to the input schema as shown in Figure 3 and at the same time changing the strategy to also pick the path from the BusRoute via Coordinator to Person. This evolution violates the hidden assumption that the advice attached to Person makes about its context; that it will be executed the first time in the context of either a Driver or a Passengers (Listing 5).

### 4. CONTROLLING ADAPTIVE PROGRAM EVOLUTION

In this section we present a language for asserting the “correctness” of adaptive programs based on the context(s) in which certain advice executes and how often it does so. Context correctness assertions can be employed for controlling evolutions leading to drastic impacts. Cardinality correctness assertions can be employed to control evolutions leading to minor impacts.

#### 4.1 Syntax

Listing 7 shows an EBNF grammar of the proposed language of assertions. An assertion annotates one advice. This way the assertion gets associated with one advised type. On the other hand, each advice can be annotated with any number of assertions. There are two types of assertions: ContextAssertion s for controlling evolutions leading to drastic impacts, and CardinalityAssertion s for controlling evolutions leading to minor impacts.

Context assertions are further split into three kinds: direct context assertions, forbidden context assertions, and required context assertions.

A direct context assertion is parameterized by a set of required types, and used to assert that the traversal visits one of these required types right before the type to which the assertion is associated (i.e., no other advised type is visited in between). A forbidden context assertion is parameterized by a set of forbidden types, and used to assert that none of these forbidden types is visited before the type to which the assertion is associated.
A required context assertions is parameterized by a set of required types and a set of sets of alternative types, and used to assert that the traversal visits all of the required types and only one type from each set of alternative types before the the type to which the assertion is associated is visited.

A cardinality assertion is parameterized by a “from” node, and used to assert that the number of ways for reaching the type to which the assertion is associated from the “from” type satisfies a certain predicate. Predicates can be formed from comparison operators and logical connectives.

4.2 Semantics

The meaning of an assertion is the set of adaptive programs for which it is defined and holds. Since the assertions we defined above are based on the runtime behavior of the adaptive program, it is enough to check them against the smooth traversal graph.

4.2.1 Checking Direct Context Assertions

A direct context assertion $c$ is associated with an advised type $c$.type and has a set of required nodes $c$.required. For checking a direct context assertion we use:

\[
\text{getLeadingTypes}(c\text{.type}) \subseteq c\text{.required}
\]

The metafunction $\text{getLeadingTypes}(t)$ goes through all smoothed traversal graph nodes labeled with $t$, and for each node, it finds the set of labels of its predecessors. Finally, the union of all these sets is returned.

4.2.2 Checking Forbidden Context Assertions

A forbidden context assertion $c$ is associated with an advised type $c$.type and has a set of forbidden nodes $c$.forbidden. For checking a forbidden context assertion we use:

\[
\forall p \in \text{getPaths}(\hat{t}, c\text{.type}) : c\text{.forbidden} \cap \text{getTypes}(p) = \emptyset
\]

The metafunction $\text{getPaths}(t_1, t_2)$ returns the set of all paths connecting the smoothed traversal graph node labeled with $t_1$ to the smoothed traversal graph node labeled with $t_2$. The carat symbol (ˆ) denotes the root of the smoothed traversal graph. Paths are represented as sets of nodes. The metafunction $\text{getTypes}(p)$ is used to retrieve the set of types mentioned in a path $p$.

4.2.3 Checking Required Context Assertions

A required context assertion $c$ is associated with an advised type $c$.type and has a set of required nodes $c$.required and a set $c$.alternatives of set of alternative types. For checking a required context assertion we use:

\[
\forall p \in \text{getPaths}(\hat{t}, c\text{.type}) : c\text{.required} \subseteq \text{getTypes}(p)
\]

\[
\land \forall a \in c\text{.alternatives} : |\text{getTypes}(p) \cap a\text{.types}| = 1
\]

4.2.4 Checking Cardinality Assertions

A cardinality assertion $c$ is associated with an advised type $c$.type, a from type $c$.from, and a predicate $c$.pred. For checking a cardinality assertion we use:

\[
c\text{.pred}(|\text{getPaths}(c\text{.from}, c\text{.type})|)
\]

5. STRICHER COMPATIBILITY NOTION

In this section we present a stricter notion of compatibility between the parts of an adaptive program. The proposed notion has four criteria: two of them are intended to eliminate conceptual mismatches between the strategy and the input schema. The other two are intended to eliminate conceptual mismatches between the strategy and the behavior. The trickiest part of defining the new notion is preserving the expressiveness of the adaptive programming paradigm.

5.1 Establishing Compatibility Between the Strategy and the Input Schema

Criterion 1: The positive core of the strategy must be identical to the smoothed input schema.

The positive core of the strategy contains the set of all desired paths. It is obtained from the strategy by dropping all bypassing constraints. Figure 10 shows an example of a strategy with a bypassing constraint.

The smoothed input schema is the graph obtained from the input schema by smoothing out all nodes that are not mentioned in the strategy. To smooth out a node $n$ from a graph, every predecessors of $n$ is connected to all of $n$’s successors. Then $n$ is removed along with all of its incident edges.

This criteria can be violated in two ways: the smoothed input schema has either fewer nodes and/or edges, or it has more edges than the positive core of the strategy.

As an example illustrating the first way of violating this criterion, consider an evolution to the input schema shown in Figure 2(a) that drops the Driver node altogether. This means that the developper who wrote the strategy “thinks” that a Person object is available from a BusRoute object instead of a Driver object, which is not true from the point of view of the input schema developer. In fact, strategies and input schemas that are incompatible in the old sense always violate the first criterion this way. This is the reason why the new compatibility criteria are stricter than the old one.

It is always possible to transform an adaptive program with this kind of conceptual mismatch into another adaptive program without this kind of conceptual mismatch while keeping the same runtime behavior. Simply, drop those strategy graph edges (and nodes) that do not appear in the smoothed input schema.

As an example illustrating the second way of violating this criterion, consider an evolution to the example shown in Figure 2 in which one of the developers decided that a bus driver can also be seen as a passenger too thus evolving the input schema to the one shown in Figure 9.

In this example, the strategy developer thinks that Driver objects are not contained in Passengers objects, which is not true according to the input schema. Eliminating this conceptual mismatch involves computing the traversal graph, smoothing out nodes not mentioned in the strategy, and finally adding bypassing constraints that bypass all of the nodes mentioned in the strategy provided that the bypassing constraints are not superfluous themselves.

Criterion 2: Every bypassed node on an edge from $S$ to $T$ in the strategy must appear on at least one path connecting $S$ to $T$ in the input schema. Furthermore, there must be at least one path connecting $S$ to $T$ in the input schema such that it does not go through any node that is bypassed on the edge from $S$ to $T$ in the strategy.

As an example illustrating the conceptual mismatch arising when this criterion is violated, suppose that the strategy shown in Figure 2(b) was restricted to the one shown in Fig-
There is an equivalent adaptive program that is not tighter coupling does not necessarily sacrifice the genericity of adaptive programs because we have already argued that for every adaptive program excluded. Also, assertions, when used wisely, can cut mostly on unsafe evolutions.

Suppose that we want to add a new advice to the behavior. This might trigger a change in the strategy graph if the newly advised type does not show up in the strategy, otherwise the third criterion will be violated. Likewise, dropping an advice from the behavior might violate the fourth criterion. Moreover, adding or removing an advice might disturb the context of other advices leading to violated assertions.

Evolving the strategy by dropping a node might violate the third criterion if the dropped a node was advised. Dropping a bypassing constraint might either: violate a cardinality assertion because a new path is picked from the input schema, or disturb the context of an existing advice because the new path acts as a short cut to the type that advice is attached to. Dropping an edge might violate the first criterion.

Evolving the strategy graph by adding a node might violate the fourth criterion if the newly added node does not lead to an advised type. Adding a bypassing constraint might violate the second criterion if it excludes all excluded paths in the input schema that connects two nodes that are connected by an edge in the strategy. Adding a bypassing constraint might also violate a cardinality constraint, disturb the context of other advices, Adding an edge to the strategy might violate a cardinality constraint, act as a short cut to an advised type thus disturbing the context, or lead to the violate the first criterion because the newly added edge does not have a corresponding path in the input schema.

Evolving the input schema by adding or edges can violate a cardinality constraints, disturb the context of an advice, or violate the first criterion. Dropping nodes or edges might also violate a cardinality constraint, disturb the context of other advices leading to violated assertions.

Evolving the cou-pling between the three parts of an adaptive program. It is now more likely that evolving one of the components can trigger a series of changes in the other components. The tighter coupling does not necessarily sacrifice the genericity of adaptive programs because we have already argued that for every adaptive program excluded by any of the four criteria, there is an equivalent adaptive program that is not 6. EVALUATION

Adopting the new notion of compatibility tightens the coupling between the three parts of an adaptive program. It is now more likely that evolving one of the components can trigger a series of changes in the other components. The tightening does not necessarily sacrifice the genericity of adaptive programs because we have already argued that for every adaptive program excluded by any of the four criteria, there is an equivalent adaptive program that is not

5.2 Establishing Compatibility Between the Strategy and the Behavior

Criterion 3: All Advised nodes must be mentioned in the strategy.

The rationale behind this criterion is that advising types that are not mentioned in the strategy graph means that the strategy is too general for the behavior. No guarantees can be made about the execution context of advice for types that are not mentioned in the strategy graph. These advice are not even guaranteed to execute in the first place. Eliminating this kind of conceptual mismatch involves computing the traversal graph, then smoothing out nodes that are neither mentioned in the strategy nor advised.

Criterion 4: Every strategy graph node is either advised or can reach an advised node.

The rationale behind this criterion is that the existence of strategy nodes that are neither advised nor lead to advised nodes means that the strategy is too specific for the behavior and can be generalized to allow more compatible input schemas by dropping those strategy nodes causing the violation.

7. RELATED WORK

The problem we addressed in this paper is similar in essence to the Fragile Pointcut Problem [17] in the Aspect Oriented Programming community. There are two main difference between the two problems: first, in AOP, the join point model contains the set of all events that occur during program execution. A pointcut expression in AOP selects a subset of join points. In our problem, the join point model contains the set of all paths in the input schema. A strategy selects a subset of these join points. Second, in AOP, the selection is name based. Whereas, in AP, the selection is mostly structure based. Still, existing approaches for solving the fragile pointcut problem (e.g., model based pointcuts [8, 18] and pointcut expression recovery [9]) can be used to complement our approach.

There are a number of generic programming technolo-

\[
\begin{align*}
\text{Criterion 4: } & \text{Every strategy graph node is either advised or can reach an advised node.} \\
\text{Criterion 3: } & \text{All Advised nodes must be mentioned in the strategy.}
\end{align*}
\]

There are two instantiations of AOP concepts in AP: The dynamic instantiation and the static instantiation. In the dynamic instantiation, the join point model is the set of all traversal events (e.g., entering/leaving a node). Method signatures serve as pointcut expressions and method bodies serve as advice. In the static instantiation, the join point model is the set of all input schema paths. The strategy serves as a pointcut expression. Pointcuts are enhanced by injecting traversal methods along them.
AP communities. In the SP community, a basic notion of compatibility is checked during the specialization of strategic programs [6, 11] provides a study of programming errors in SP, proposes refinements to SP which resembles our stricter notion of compatibility. In the AP community, Demeter Interfaces [16] focuses on establishing compatibility between the input schema and the constraints imposed by the traversal strategy. The DemeterF [1] type system [4] focuses on establishing compatibility between the traversal strategy and the set of collaborating methods comprising the behavior.

8. CONCLUSION AND FUTURE WORK

In this paper we presented a study of adaptive program evolution and two complementary approaches for controlling unsafe evolution of adaptive programs. We plan to introduce adaptive programming as a technology for event based processing of XML [2].

9. ACKNOWLEDGMENTS

We would like to thank Bryan Chadwick and our anonymous reviewers for all their valuable feedback. This work is supported in part by Grantham Mayo Van Otterloo, LLC.

10. REFERENCES

[1] DemeterF. 
http://www.ccs.neu.edu/home/chadwick/demeterf/


11. REFERENCES

[1] DemeterF.
http://www.ccs.neu.edu/home/chadwick/demeterf/


