Controlled Evolution of Adaptive Programs

Ahmed Abdelmeged Therapon Skotiniotis Karl Lieberherr

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{mohsen,skotthe,lieber}@ccs.neu.edu

ABSTRACT

Adaptive programming (AP) is a programming paradigm for expressing computations over semi-structured data graphs. An adaptive program is written against an entire family of input schemas and, at compilation time, it is *instantiated* for a specific input schema. Adaptive programs seamlessly *adapt* to switching of their input schemas. Due to their organization as traversals, adaptive programs are also retargetabe to different execution platforms (e.g. single and multi-core architectures).

As other programming paradigms, adaptive programs are also 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 unsafe evolutions.

Categories and Subject Descriptors

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

General Terms

Languages, Design, Reliability

Keywords

Adaptive Programming, Evolution, Assertion

1. INTRODUCTION

Adaptive programming (AP) is a programming paradigm for expressing computations over semi-structured data graphs [7]. An adaptive program is written against an entire family of input schemas and, at compilation time, it is *instantiated* for a specific input schema. Adaptive programs seamlessly *adapt* to switching of their input schemas. Due to their organization as traversals, adaptive programs are also retargetabe to different execution platforms [2] (e.g. single and multi-core architectures).

An adaptive program receives a data graph as input and comprises two parts: a strategy, and a behavior. The strategy is a regular-expression-like specification that serves two purposes: guiding a traversal of the input data graph and, serving as an interface for an entire family of input schemas against which the behavior is written. The strategy does not have to fully specify every single detail of the traversal. This is the reason why adaptive programs seamlessly adapt to evolutions of their input schema [8]. The behavior is a set of collaborating aspects that advise the traversal. Each aspect attaches an advice to a specific type of objects and advices are executed when the traversal reaches an object with the type they are attached to.

An adaptive program instance is a specialization of an adaptive program using for a specific schema, which defines the structure of the input data graph. The input schema is also employed to optimize the runtime overhead of the traversal. Throughout the rest of the paper, we shall use the term adaptive program to refer to an adaptive program instance.

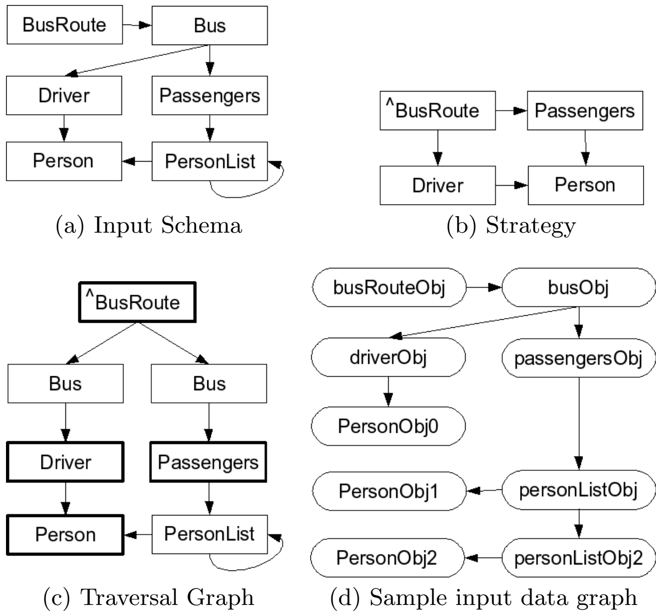
1.1 Background

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. We want to find the number of people, including the driver, on board of the bus.

Figure 1(a) shows the schema for the input data graph shown in Figure 1(d). Figure 1(b) shows the strategy which says: "go from a `BusRoute` object to a `Person` object via either a `Driver` object or a `Passengers` object"[1]. Listing 1 shows the behavior, which contains two aspects: one is attached to `BusRoute`, which initializes a global counter, and the other is attached to `Person`, and 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 *Traver-*

[1]The carat symbol ($\wedge$) is used to denote the source node of a strategy or a traversal graph



(a) Input Schema (b) Strategy

(c) Traversal Graph (d) Sample input data graph

Figure 1: Bus Route Example

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

Listing 1: Counting People on Board

sal Graph; Figure 1(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 in the strategy.

To execute our adaptive program on the data graph shown in Figure 1(d) (which satisfies the input schema) we need both the traversal graph and the behavior. 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 children 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`.

```
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); } } }

GetNextTgNodes(currentTgNodes, label)
{ nextTgNodes = EmptyList;
  foreach(tgNode:currentTgNodes)
  { nextTgNodes.append(
        tgNode.getChildren(label)); }
  return nextTGNodes; }
```

Listing 2: Adaptive Program Execution

In case the set of 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 2 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 2 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.

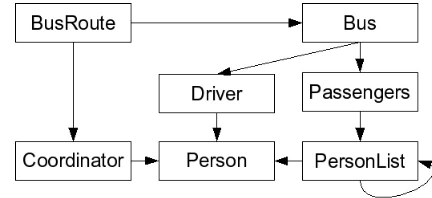


Figure 2: Bus Route Schema with a Coordinator

Another change to the input schema that our adaptive program can seamlessly handle, is to add an `Operator` node between `Bus` and `Driver`. Figure 3 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[2]. The reason is that `Operator` is not advised.

1.2 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 4(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

[2]Judging the runtime behavior of an adaptive program only by the trace of advice execution events.

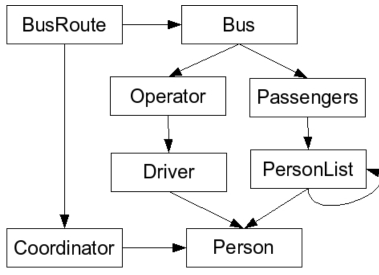


Figure 3: Bus Route Schema with a Coordinator and an Operator

programs [3]. The dashed circle contains all "correct" adaptive programs for some application specific correctness criteria that the adaptive programming system might be unaware of.

The arrows represent various kinds of evolutions of adaptive programs. Starting with a "correct" 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*.

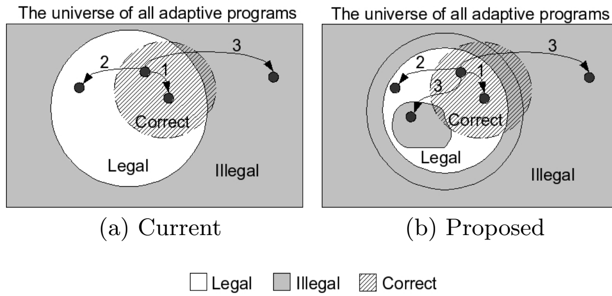


Figure 4: Evolution of Adaptive Programs

1.3 Problem

The first problem that we tackle in this paper is that 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 therefore, unsafe evolutions can go uncaught. This is a problem of virtually every programming paradigm 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. For example, even though the strategy defines the interface for the legal family of input schemas, the behavior can still advise nodes that are not mentioned in the strategy and are not even guaranteed to exist in the traversal graph.

[3]An adaptive program is legal if its input schema is *compatible* with its strategy. That is for every edge in the strategy connecting a node `S` to another node `T`, `T` is *reachable* from `S` in the input schema. The restriction of legal adaptive to those with *compatible* strategy and input schema does not jeopardize the expressiveness of adaptive programs because an adaptive program with incompatible strategy and input schema can always be transformed into a legal adaptive program. Simply by removing all strategy edges whose target is not reachable from the edge source in the input schema.

1.4 Contributions

This paper aims at controlling the unsafe evolutions of adaptive programs. To that end, we propose that developers should declare a "correctness criteria" for 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 4(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.

We also propose a second complementary approach for controlling the unsafe evolution of adaptive programs which is based on incorporating a stricter notion of *compatibility* that encompasses all three components of an adaptive program and excludes those adaptive programs that contain some form of a "conceptual mismatch" regardless of their "correctness". The tricky part here is preserving the expressiveness of the adaptive programming paradigm. Figure 4(b) 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 the stricter notion of compatibility.

1.5 Organization

In section 2 we present a *comprehensive* study of the evolution of adaptive programs with the goal of identifying unsafe evolutions. In section 3 we present a language for declaring "approximate correctness criteria". In section 4 we present

our stricter notion of compatibility. In section ?? we present some of the related work. Section 6 concludes the paper.

2. EVOLUTION OF ADAPTIVE PROGRAMS

In this section we study the effect of evolution on the runtime behavior of adaptive programs. However, Listing 2 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.

2.1 Runtime Behavior Representation

Two key observations can be made from Listing 2 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. Moreover, 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 two nodes: one labeled `S` and the second labeled `T` and the second is reachable from the first. 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 in 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[4]. 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 bus

[4]This is actually a representation of a *family* of paths rather than single paths. A loop in our representation corresponds to an infinite number of paths. Every node along the path must be the source of only one forward edge and any number of backward edges

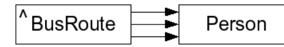
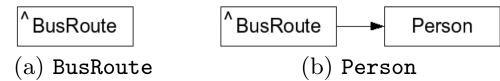


Figure 5: The Smoothed Traversal Graph

route example mentioned in the introduction. 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`)} and { (`BusRoute`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `Person`)} and { (`BusRoute`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `PersonList`), (`PersonList`, `Person`)}.



(a) `BusRoute` (b) `Person`

Figure 6: Advice Execution Contexts

Moreover, 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).

2.2 Impacts of Evolution on Adaptive programs

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

2.2.1 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 ending 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.

2.2.2 *Minor Impact*

Suppose that we were to allow many buses on the same route. The evolved input schema is shown in Figure 8(a); an unadvised node `BusList` is inserted. However, this node 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 the evolution there was only one way to do so.

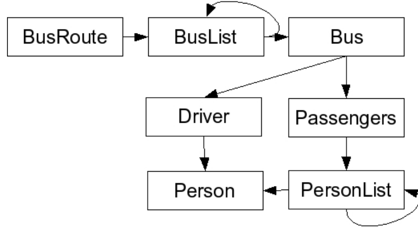
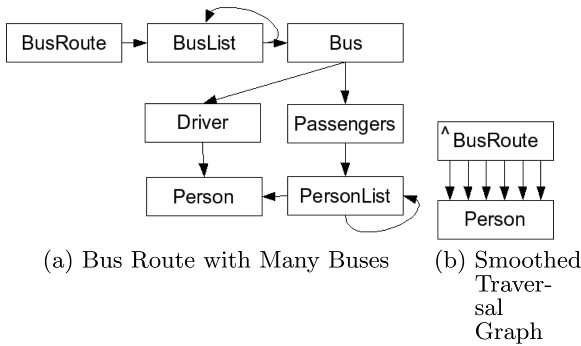


Figure 7: Bus Route with Many Buses



(a) Bus Route with Many Buses (b) Smoothed Traversal Graph

Figure 8: Bus Route with Many Busses

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` (see 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 advices 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 8(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`)} and { (`BusRoute`, `BusList`), (`BusList`, `BusList`), (`BusList`, `Bus`), (`Bus`, `Driver`), (`Driver`, `Person`)} and { (`BusRoute`, `BusList`), (`BusList`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `Person`)} and { (`BusRoute`, `BusList`), (`BusList`, `BusList`), (`BusList`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `Person`)} and { (`BusRoute`, `BusList`), (`BusList`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `PersonList`), (`PersonList`, `Person`)} and { (`BusRoute`, `BusList`), (`BusList`, `BusList`), (`BusList`, `Bus`), (`Bus`, `Passengers`), (`Passengers`, `PersonList`), (`PersonList`, `PersonList`), (`PersonList`, `Person`)}

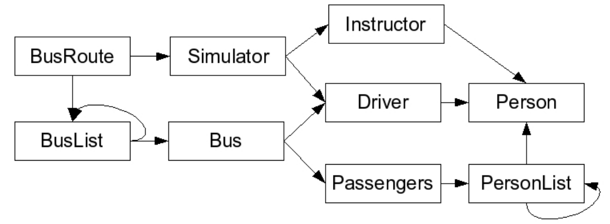


Figure 9: Bus Route with a Simulator

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` change as the number of ways of going from a `BusRoute` to a `Driver` change.

In general, an evolution that results in a change to the number of ways that a node can lead to another node in the smoothed traversal graph without changing the any 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, in 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 on the smoothed traversal graph need to be controlled.

2.2.3 *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, suppose that we need to keep two separate counts: the number of passengers on board of any bus as well as a the number of drivers on duty. Listing 3 shows the behavior we use to keep the two counts. The set of advised nodes in the new behavior include `Passenger`, `Driver`, and `Person` meaning that nodes labeled with these three types will *not* be smoothed out in the smoothed traversal graph.

As an example of an unsafe evolution leading to a drastic impact, consider adding a `Coordinator` to the input schema as shown in Figure 2, 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

```
class CountPassengersAndDrivers
{ private int noOfPassengersOnBoard = 0;
  private int noOfDriversOnDuty = 0;
  private boolean driverSeen;
  private boolean passengerSeen;

  before(Driver driver)
   { driverSeen = true;
     passengerSeen = false; }
  before(Passenger passenger)
   { passengerSeen = true;
     driverSeen = false; }
  before(Person person)
   { if (driverSeen)
     { noOfDriversOnDuty +=1;
       driverSeen = false; }
     if (passengerSeen)
     { noOfPassengersOnBoard +=1;
       passengerSeen = false; } } }
```

Listing 3: Counting Passengers and Drivers

```
Assertion = "execute" ( Context | Cardinality )
Context = Direct | Forbidden | Required
Direct = "directly" "in" AdvisedType*
Forbidden = "not" "in" AdvisedType*
Required = "in" AdvisedType*
                AlternativeTypes*
AlternativeTypes = "[" AdvisedType* "]"
AdvisedType = IDENTIFIER

Cardinality = Predicate "in" AdvisedType
Predicate = Simple | Composite
Composite = "(" Predicate Op Prediate ")"
Simple = Rel INTEGER
Op = ( "and" | "or" )
Rel = ">" | "<" | "==" | "<=" | ">=" | "!="
```

Listing 4: Language for Asserting the Correctness of Adaptive Programs

context of either a `Driver` or a `Passengers` (see Listing 3).

3. 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.

3.1 Syntax

Listing 4 shows the 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` for controlling evolutions leading to drastic impacts, and `CardinalityAssertion` 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 assertions is parameterized by a set of required types, and used to assert that the traversal visits one 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 assertions 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.

3.2 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 smoothed traversal graph.

3.2.1 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:

$$getLeadingTypes(c.type) \subseteq c.required$$

The metafunction *getLeadingTypes* goes through the all smoothed traversal graph nodes labeled with *c.type*, and for each node, it finds the set of labels of its predecessors. Finally, the union of all these sets is returned.

3.2.2 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:

$$\forall p \in getPaths(\wedge, c.type) : c.forbidden \cap getTypes(p) = \emptyset$$

The metafunction $getPaths(t1, t2)$ is used to retrieve the set of all paths connecting the smoothed traversal graph node labeled $t1$ to the smoothed traversal graph node labeled $t2$. $\wedge$ denotes the root of the smoothed traversal graph. Paths are represented as sets of edges. the metafunction $getTypes(p)$ is used to retrieve the set of types mentioned in a path p.

3.2.3 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:

$$\forall p \in getPaths(\wedge, c.type) : c.required \subseteq getTypes(p)$$
$$\bigwedge \forall a \in c.alternatives : |getTypes(p) \cap a.types| = 1$$

3.2.4 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:

$$c.pred(|getPaths(c.from, c.type)|)$$

4. STRICTER LEGALITY NOTION

In this section we present a stricter notion of legal adaptive programs. In addition to satisfying all assertions, the proposed notion has five more criteria: Three 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. We conclude this section with a discussion of the ramification of the new notion on the evolution of adaptive programs.

4.1 Establishing Compatibility Between the Strategy and the Input Schema

The first kind of conceptual mismatch is: the absence of desired input schema paths.

As an example, consider an evolution to the input schema shown in Figure 1(a) that drops the `Driver` node altogether. This means that the developer who wrote the strategy "thinks" that a `Person` object is accessible from a `BusRoute` object through a `Driver` object, which is not true from the point of view of the input schema developer. This is in fact the first and only legality criteria for adaptive programs [6].

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 with the same runtime behavior. Simply, by dropping those strategy graph edges (and nodes) that do not have a corresponding path in the input schema.

The second kind of conceptual mismatch is: the absence of undesired input schema paths.

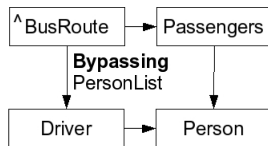


Figure 10: Restricting the Strategy Shown in Figure 1(b)

As an example, suppose that the strategy shown in Figure 1(b) was restricted to the one shown in Figure 10 by adding a `Bypassing PersonList` constraint to the edge connecting `Driver` to `Person` meaning that we are interested in `Person` objects reachable from `Driver` objects but not having any `PersonList` objects on the way.

In this example, the strategy developer thinks that there might be a `PersonList` on the way from `Driver` to `Person`, which is not true according to the input schema. This conceptual mismatch can be eliminated by dropping the redundant bypassing constraint.

The third kind of conceptual mismatch is: the existence of neither desired nor undesired input schema paths connecting strategy nodes.

As an example, consider an evolution 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 11.

In this example, the strategy developer thinks that there is `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 redundant themselves

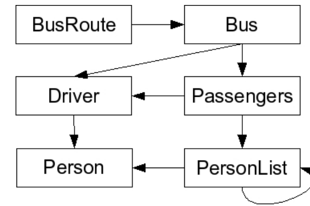


Figure 11: Bus Route with Drivers as Passengers

4.2 Establishing Compatibility Between the Strategy and the Behavior

The fourth kind of conceptual mismatch is: advising types that are not mentioned in the strategy graph. This means that the strategy is too general for the behavior. No guarantees can be made about the context in which advices for types not mentioned in the strategy graph. These advices 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.

The fifth kind of conceptual mismatch is: the existence of irrelevant strategy nodes. A strategy node is irrelevant if it is neither labeled with an advised type nor can reach an advised type. This simply means that the strategy is too specific for the behavior and can be generalized to allow more compatible input schemas.

4.3 Ramifications on Evolution

Adopting the new notion of legality *tightens* the coupling between the three components 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 assertions, when used wisely, can cut mostly on unsafe evolutions, and we have already argued that the other five criteria also 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 fourth kind of conceptual mismatch will occur. Likewise, dropping an advice from the behavior might trigger the fifth kind of conceptual mismatch. Moreover, adding or removing an advice might *disturb* the context of other advices.

Evolving the strategy by dropping a node might result in the fourth kind of conceptual mismatch if the dropped node was advised. Dropping a bypassing constraint might either: violate a cardinality assertion because a new path is picked from the input schema, disturb the context of an existing advice because the new path acts as a short cut to the type that advice is attached to, or result in the third kind of conceptual mismatch if the bypassed type exists elsewhere in the strategy. Dropping an edge will result in the third kind of conceptual mismatch if its source and target nodes remain in the strategy.

Evolving the strategy graph by adding a node might result in the fifth kind of conceptual mismatch if it does not lead to an advised type. Adding a bypassing constraint might result in the first kind of conceptual mismatch if it excludes the only path in the input schema that conceits 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, or result in the second kind of conceptual mismatch. 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 first kind of conceptual mismatch because the newly added edge does not have a corresponding path in the input schema.

Evolving the input schema by adding non advised nodes or edges can violate a cardinality constraints, disturb the context of an advice, or lead to the third kind of conceptual mismatches. Dropping nodes or edges might also violate a cardinality constraint, disturb the context of an advice, or lead to the first kind of conceptual mismatches.

5. RELATED WORK

There is a number of generic programming technologies [5, 4]. To the best of our knowledge, there has been no work related to controlled evolution in these communities. The most relevant related work is the work on Demeter interfaces in [9], and the work on DemeterF type system [3].

6. CONCLUSION AND FUTURE WORK

In this paper we presented a study of adaptive program evolution and two approaches for controlling unsafe evolution of adaptive programs. We plan to introduce adaptive programming as a technology for event based processing of XML [1].

7. ACKNOWLEDGMENTS

This work is supported in part by Grantham Mayo Van Otterloo, LLC.

8. REFERENCES

[1] *The SAX Project*. `http://www.saxproject.org/`.

[2] Bryan Chadwick and Karl Lieberherr. Functional Adaptive Programming. Technical Report NU-CCIS-08-75, CCIS/PRL, Northeastern University, Boston, October 2008.

[3] Bryan Chadwick and Karl Lieberherr. A type system for functional traversal-based aspects. In *FOAL '09: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 1–6, New York, NY, USA, 2009. ACM.

[4] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.

[5] Ralf Lämmel, Eelco Visser, and Joost Visser. Strategic programming meets adaptive programming. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 168–177, New York, NY, USA, 2003. ACM.

[6] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[7] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, web book at www.ccs.neu.edu/research/demeter.

[8] Johan Ovlinger and Mitchell Wand. A language for specifying recursive traversals of object structures. *SIGPLAN Not.*, 34(10):70–81, 1999.

[9] Therapon Skotiniotis, Jeffrey Palm, and Karl J. Lieberherr. Demeter interfaces: Adaptive programming without surprises. In *European Conference on Object-Oriented Programming*, pages 477–500, Nantes, France, 2006. Springer Verlag Lecture Notes.