Poplar: Java Composition with Labels and AI Planning

Johan Nyström-Persson
University of Tokyo and National Institute of Informatics
Tokyo, Japan
jtnystrom@is.s.u-tokyo.ac.jp

Shinichi Honiden
University of Tokyo and National Institute of Informatics
Tokyo, Japan
honiden@nii.ac.jp

ABSTRACT

Class evolution in object-oriented programming often causes so-called breaking changes, largely because of the rigidity of component interconnections in the form of explicit method calls and field accesses. We present a Java extension, Poplar, which we are currently developing. In Poplar, inter-component dependencies are expressed using declarative queries; concrete linking code, generated using a planning algorithm, replaces these at compile time. We show how Poplar can enable fully automatic integration of Java components through evolvable and statically checkable integration links, pointing the way to a new general composition method for object-oriented languages.

Categories and Subject Descriptors

D3.3 [**Programming Languages**]: Frameworks
; D3.3 [**Programming Languages**]: Constraints

General Terms

Languages

Keywords

Components, protocols, code synthesis, object-oriented programming, AI planning, adaptation, evolution, composition

1. INTRODUCTION

Two essential and related properties of object-oriented programming languages like Java are encapsulation and polymorphism. Encapsulation is the principle of separating interface from implementation, and this in turn enables polymorphism, whereby the runtime type of an object may be different from its declared type in the source code, and thus unknown to the caller. When two classes have the same interfaces, according to the principle of behavioural subtyping[9], the implementations should be substitutable for each other and all expected safety properties should be retained. Contemporary programming paradigms such as component-based software development (CBSD) [15, 14] draw heavily on these principles, and in principle, all object-oriented programming languages strive to simplify code reuse. In what follows, we refer to components in the most general sense of the term, as sets of classes.

Theoretically, interfaces of classes should only change in backward-compatible ways once they have been published, for instance through the addition of new methods, or through the widening of assumed preconditions and narrowing of assumed postconditions. Interface changes that require client classes to update their associated client code are called breaking changes. While developers strive not to make such breaking changes, it has been found that in practice they are commonplace [4]. Breaking changes introduce a large cost into component-based software development, since potentially every dependent class may have to be updated. In other words, CBSD, as it is practised today, suffers from a conflict between software evolution and flexibility of composition.

The following are some examples of language-level breaking changes that can occur in modern imperative object-oriented languages.

Name changes. The renaming of a method, everything else being the same.

Protocol changes. Often, a sequence of method invocations is required to establish a certain effect or compute a certain value. When this sequence changes from one class version to another, we say that a protocol change has occurred. This includes permutations of protocol steps, but also addition of new steps and removal of old steps.

Type changes. Methods may be moved from one class to another; argument and return types may be changed to incompatible types.

Signature changes. The number of arguments that methods require may change, without visibly affecting the functionality that existing clients receive.

In addition, there are changes that occur above the level of the language, such as conceptual semantic changes and quality attributes [1]."Method calls [and field accesses] are the assembly language of software interconnection" [12], and this situation has several problems. We are currently developing a Java extension, Poplar, which adds several concepts to the Java language to support a new kind of composition methodology, in which integration requests are expressed using declarative *queries*. At compile time, we generate fragments of concrete linking code, called *solutions* to the queries.

Poplar associates a set of labels with each variable and describes interfaces in terms of preconditions, postconditions and invariants of fields, parameters, return values and method receivers. Labels are a compile-time property which are used to allow the generation of solutions by a search algorithm. The implementation that we are constructing uses the Partial Order Planning (POP) algorithm, but in principle many different algorithms can be used. Our design permits both modular integration and modular checking of integrations. We intend for such integrations to be used across component boundaries, where developers can anticipate that interfaces may change in the future, but it is also possible to retrofit them into an existing code base.

In what follows, we give an example of the application of Poplar to the Java time and date API (Section 2), and describe our use of the Partial Order Planning algorithm (Section 3). We then give some general remarks on our approach (Section 4), discuss related work (Section 5) and conclude (Section 6).

2. INTEGRATING TIME AND DATE CODE

We introduce the Poplar Java integration mechanism using a simple example from the real world. The time and date API of the standard Java libraries changed substantially between version 1.4 and version 1.5 of the language. In version 1.4, the following code was used to obtain the current hour of the day:

```
Date now = new Date();
int hour = now.getHour();
```

In Java 1.5 and later versions, the following code is used:

```
Calendar now = Calendar.getCalendar();
int hour = now.get(Calendar.HOUR_OF_DAY);
```

Even though the Java 1.5 libraries keep the old version of the API, this is representative of a breaking change that may occur in practice, and API publishers generally prefer not to have to preserve old versions.

In principle, Poplar considers integration sites to have one of two possible purposes: producing values or producing effects. Clearly, in this case, a client component that wants to use the time and date API wants to do the former. Before we can request the production of a value, we must annotate the API that is provided by the service component. In the case of Java 1.4, the component supplier should provide annotations similar to the ones in Figure 1.

2.1 Labels

We have added the labels annotation as a new member of classes and interfaces. In the case of the interface TimeAndDate, labels are provided for the int type using the notation labels(int). Once these labels have been defined, we can logically distinguish between integers that have these labels and integers that do not, as a lightweight refinement of the type system. Since the Date class implements TimeAndDate, references to nowHour in this class are understood to refer to the label

```
interface TimeAndDate {
  labels(int) nowHour, nowMinute, nowSecond;
}

class Date implements TimeAndDate {
  labels currentDate;
  Date()
    result: +currentDate;
  int getHour()
    this: currentDate,
    result: + nowHour;
  /* Similar annotations for getMinute(),
       getSecond(), etc. */
}
```

Figure 1: TimeAndDate annotations for Java 1.4.

defined in the TimeAndDate interface, but it is possible for other interfaces to define labels with the same name, and their meaning might be different. Disambiguation should be done in the usual way using fully qualified names where necessary.

In the Date class we have added pre- and postconditions to the methods. The constructor Date() declares that the result, ie the return value, will have the new label currentDate. This label was declared in the Date class itself. The + sign indicates that a new label is added. In contrast, the getHour method indicates that for the this variable, the receiver of the method, an invariant of the currentDate label is expected - the label must be owned by the this object prior to method invocation, and it will remain after the invocation. When this method is invoked, the return value will be an integer which has the nowHour label. Here, the labels describe one kind of useful application of the method, but not mandatory constraints on it. As for the client component which wants to produce the value corresponding to the current hour of the day, its code should resemble the following:

```
class TimeUtils implements TimeAndDate {
  void printHour() {
    int hour = #produce(int, nowHour);
    System.out.println("The current hour is: "
        + hour);
  }
}
```

Again, we use the TimeAndDate interface to refer to the nowHour label. We request a production of an integer value with this label using the #produce query. We prefix queries with a '#' sign to distinguish them from normal Java code. At compile time, the Poplar solver will find a solution to this query and replace it with a sequence of Java statements. Such statements can be field accesses or method invocations, including constructor invocations. Code resulting from #produce queries will return a single value, and the queries may thus be "assigned" to a variable, as in the example we have just shown.

The solver uses a planning algorithm to find a solution to the query. The plan search will proceed backwards from the goal to the assumptions. In this case, the goal is the existence of a variable of type int and with label nowHour. First the planner needs to find all actions that can produce such a variable (method invocations and field accesses). If the only one available is the one we declared above (Date.getHour), then once this method has been selected, a new set of precon-

```
Class Calendar implements TimeAndDate {
  labels(int) hourMarker, minuteMarker,
      secondMarker;
  labels defaultTimeZone;

  final int HOUR_OF_DAY:(hourMarker) = 11;

  Calendar()
    result: +defaultTimeZone { ... }
  int get(int selector)
    this: defaultTimeZone,
    (selector: hourMarker,
    result: +nowHour)?,
    (selector: minuteMarker,
    result: +nowMinute)? { ... }
}
```

Figure 2: TimeAndDate annotations for Java 1.5.

ditions will result - in order for that method to be invoked, we need to have a Date object with the currentDate label. We repeat the search and find that there is a constructor that takes no arguments and that produces such an object. This yields a complete solution, and thus, after the code has been generated and the substitution has taken place, the client class will look like the following:

```
class TimeUtils implements TimeAndDate {
  void printHour() {
    Date v1 = new Date();
    int hour = v1.getHour();
    System.out.println("The current hour is: "
        + hour);
  }
}
```

In addition, the solver will remove non-Java elements from the code, such as the label declarations from the various classes, so that the result is valid Java source code.

2.2 Upgrading to Java 1.5

Let us now consider how we could adapt this client to the Java 1.5 version. In this case, the service component would resemble the one shown in Figure 2.

In this case, all the values are accessed through one method, which takes a selector argument. We have given the field HOUR_OF_DAY an explicit label hourMarker, which links it to its possible use as an argument for the get(int) method. We group invariants and postconditions as a disjunction of conjunctions using the (a, b, ...)? syntax, which makes the pre- and postconditions inside the group optional. The plan search now takes the same query as a starting point, but the APIs supplied as input are different (and perhaps the 1.5 API is marked as taking precedence over the 1.4 one if both are available) and after substituting a solution for the query we end up with:

```
class TimeUtils implements TimeAndDate {
  void printHour() {
    Date v1 = new Calendar();
    int v2 = Calendar.HOUR_OF_DAY;
    int hour = v1.get(v2);
    System.out.println("The current hour is: "
        + hour);
  }
}
```

We show the plans visually in Figure 3. The resulting code is extracted directly from the plans. Rounded boxes are pre- and postconditions (the existence of a variable with a given label), and square boxes are actions such as method invocation and field access. Dashed lines represent sequential constraints, which impose an ordering on the actions. These constraints will be at least as strong as the dataflow dependencies of the solution, and possibly stronger due to possible conflicts.

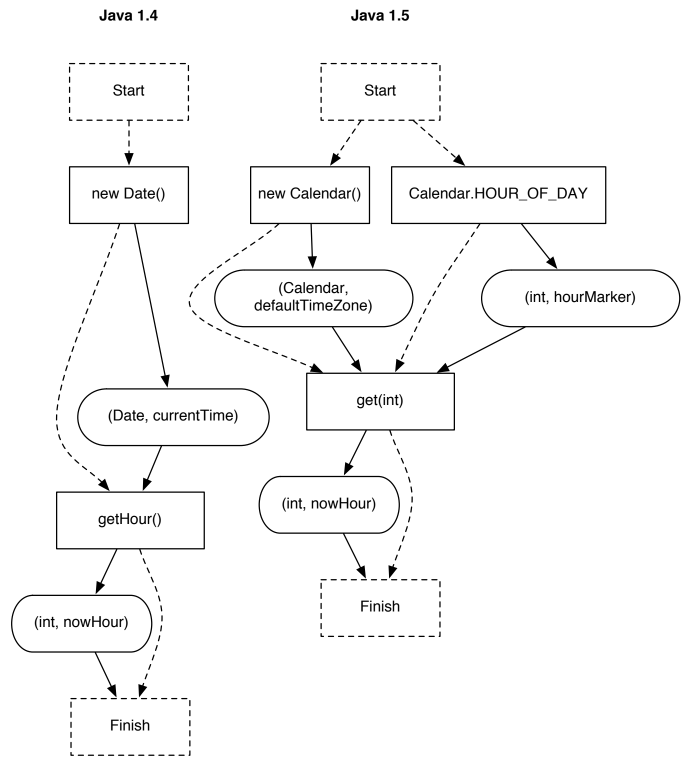

Figure 3: Visual representations of generated plans in Java 1.4 and 1.5, respectively.

These examples demonstrate how clients can automatically be reconfigured to use a new version of an API, or even a different API, given that the necessary annotations are present on both the client and the service side. In this case it would simply be a matter of re-running the integration tool with the newest service components added to the classpath.

Labels correspond to the ability of variables to participate in a given use case for an API. In the example shown here, they simply classify values according to what is being represented. A specification such as this: defaultTimeZone, (selector: hourMarker, result: +nowHour)? signifies that, assuming that the receiver is a defaultTimeZone, and that selector is an hourMarker, the return value will be able to participate in API usages that require a nowHour integer. This specification was derived in a straightforward manner from the Javadoc API documentation. However, in the full version of Poplar, which also constrains pointer aliasing, it is sometimes necessary to have access to either detailed documentation or the full source code of service components.

3. PLANNING AND CODE GENERATION

The design of Poplar does not restrict the choice of planning or search algorithm that is to be used for the code generation, but in early experiments on a prototype, we have found Partial Order Planning (POP) to be a useful algorithm.

POP gradually refines a partial ordering of some set of actions. In principle, it searches the space of all possible plans. Java statements are already in some sense partially ordered, for instance through dataflow dependencies. POP is also relatively easy for humans to understand and reason about, which may be valuable if there is a need to tweak annotations. The basic idea of the POP algorithm is that it gradually strengthens an ordering of actions, inserting causal links (connecting post- and preconditions of related actions) and new actions as necessary while maintaining a set of open preconditions. Conditions will be either of the form new(T, l), indicating a new variable of a given type and label, or label(x, l), indicating that a variable has a certain label. Note that the full version of Poplar also allows labels to be erased: code fragments can satisfy certain goals while undoing others.

A backward search from the goal towards the initial condition is performed as follows.

1. Initialise the plan to have two pseudo-actions *start* and *finish*. The effects of the start action are identical to the assumed environment of the plan, i.e. the starting conditions. The preconditions of the finish action are identical to the goals of the plan.

2. If there are no open preconditions, stop. A solution has been found.

3. Select an open precondition in the current plan.

4. For all available actions that achieve the precondition and are either already in the plan or not in the plan, create a successor plan with this action added. Also add ordering constraints and causality links for the new action.

5. For all the successors, resolve any conflicts among the causality links that might have arisen by strengthening the ordering constraints. If this is not possible, discard the successor.

6. Recurse on each successor plan. Go to step 2.

Two heuristics are needed: one for selecting the next open precondition to plan for, and one for selecting the most appropriate action to attempt for a given precondition, if several are available. In our prototype, we give priority to preconditions that either have no available actions to realise them (a fail-fast strategy) or that have only one available action (to lock in necessary decisions early). Generally, we favour preconditions with the smallest amount of available satisfying actions. We have also experimented with schemes that favour syntactic locality by preferring fields in the same class over invoking a method, preferring local methods over methods in other classes, and so on.

4. DISCUSSION

Due to space constraints, we have omitted many features of Poplar here. For instance, we classify pointers according to whether they may be aliased or not, and we have also developed a kind of label that may be established and erased as a result of program execution, unlike the immutable labels shown here. We use an effect system to reason about changes in label sets over time. This means that we can reason about methods both in terms of an upper bound on their destructive side effects, and in terms of a lower bound on their useful side effects.

The approach we have presented allows us not only to perform integrations, but also to verify that new versions of a component are valid with respect to the existing client code. When we generate code, we can also output a set of *integration assumptions* for each query that we solve. This information can be stored as an additional attribute in Java class files, for instance. When new service-side components are published, they can be checked against this information to verify whether they are valid without a recompilation. If they are not, we may attempt to find new solutions for those queries that are incompatible with the new API.

We are not simply moving the integration problem to a higher level of abstraction. Once Poplar labels have been added to an API, integrations can be performed if and only if the desired output labels, which are requested in client queries, can be produced by putting together some combination of methods and fields, given the input labels made available by the client. This allows for a lot of structural evolution in APIs; changes such as method renaming and moving methods to different classes are fully or almost fully transparent (depending on the heuristics being used).

An important limitation is that control flow constructs are not generated. For instance, we do not generate loops or if-statements. Users must deal with this manually when needed. For an iterator, separate queries could be used for the truth condition and for the loop body.

Developers might be concerned about the fact that generated code can vary from time to time. In order to protect against unwanted effects, variables with the same type and labels must be truly equivalent. If they are not substitutable for each other, more labels should be added to disambiguate.

In adopting Poplar, it is necessary to add annotations to new or existing Java code. It should be possible to make a tool that infers some or all of this information, given minimal annotations as a starting point. For instance, in [2], the authors prove the correctness of an effect inference algorithm for the Boyland-Greenhouse effect system, a modification of which is used in full Poplar. It is also possible to infer protocols from a code base, as in [10]. In addition, when Poplar is introduced into an existing code base, it should be possible to start with a small number of queries and annotations, and gradually expand the use of Poplar within the code base.

5. RELATED WORK

One of the first descriptions of the partial order planning algorithm was given by Allister and Rosenblatt [11].

There are many approaches to component integration through code synthesis. Haack [5] has created a system that unifies software components in the ML language by generating code from uninterpreted, atomic annotations, much like Poplar. However, given the differences between ML and Java, and that ML unification is the central technique in Haack's work, it is not clear if the findings can be applied to an imperative object-oriented setting.

Ireland and Stark [6] have designed a system that combines proof plans and partial order planning to generate small imperative programs. The heuristics and proof critics used here are adopted from the literature on Structured Programming, which describes principles that are to be used for manual goal-directed programming. This should be a valuable avenue for future investigation of heuristics for Poplar.

Evolution and adaptation is a well studied problem. Dig [4] has carried out an empirical study of component evolution. Strikingly, he found that between 81% and 100% of all breaking changes in several large systems were due to refactorings. Vasa et al [16] provide a high level quantitative picture of software evolution.

Zaremski and Wing [17] have studied component discovery and matching. However, as has been argued, by Kell [8, 7], for instance, composition is often not just a problem of discovering a match but also a problem of assembling and adapting what has been found.

Our approach has similarities with typestate checking. Typestate was first introduced by Strom and Yemini [13] as an approach to checking valid interactions with primitive types and simple data structures. Deline and Fähndrich [3] describe how typestate checking can be applied to a core subset of C#.

Mandelin et al. [10] describe a tool that mines typestate-like protocols from code bases, something that could be useful in generating initial Poplar annotations. Their protocols are not declared explicitly; instead, a sequence of method invocations is presumed to be acceptable when it is found in a code base that is taken to be correct. Also, their tool is designed to assist the programmer in interactive use, rather than as a language extension.

6. CONCLUSION AND FUTURE WORK

We have presented Poplar, a Java extension designed for automated component integration based on queries and code generation. By annotating individual variables in interfaces, we describe components in such a way that integration links can be generated, checked and re-integrated in a flexible fashion. We believe that the integration approach presented here could be an important step towards greater reusability and ease of maintenance in component-based development.

In future work, most importantly, we aim to provide a practical implementation and investigate how Poplar works in practice. We are also working on a formalisation of Poplar, based on MJ [2].

Acknowledgements

The authors would like to thank Liyang Hu, Zhenjiang Hu, Atsushi Igarashi, Michael Nett, Alexandre Pichot and the Honiden laboratory's MALACS group for helpful comments.

7. REFERENCES

[1] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an engineering approach to component adaptation. In *Lecture Notes in Computer Science*, volume 3938, pages 193–215. Springer-Verlag, 2006.

[2] G. Bierman, M. Parkinson, and A. Pitts. Mj: An imperative core calculus for java and java with effects. Technical Report 563, University of Cambridge, 2003.

[3] R. DeLine and M. Fähndrich. Typestates for Objects. In *Lecture Notes in Computer Science*, pages 465–490. Springer-Verlag, 2004.

[4] D. Dig and R. Johnson. The Role of Refactorings in API Evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 389–398. IEEE, 2005.

[5] C. Haack, B. Howard, A. Stoughton, and J. B. Wells. Fully Automatic Adaptation of Software Components Based on Semantic Specifications. In *AMAST '02: Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, pages 83–98, 2002.

[6] A. Ireland and J. Stark. Combining proof plans with partial order planning for imperative program synthesis. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, volume 13, pages 65–105. Springer, 2006.

[7] S. Kell. Rethinking software connectors. In *International workshop on Synthesis and analysis of component connectors in conjunction with the 6th ESEC/FSE joint meeting - SYANCO '07*, pages 1–12, New York, NY, USA, 2007. ACM.

[8] S. Kell. The Mythical Matched Modules. In *OOPSLA '09: Proceedings of the 24th annual ACM SIGPLAN Conference on Object-Oriented Programming: Systems, Languages and Applications*, pages 881–888, New York, NY, USA, 2009. ACM.

[9] B. H. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 1994.

[10] D. Mandelin, L. Xu, D. Kimelman, and R. Bodík. Jungloid Mining: Helping to Navigate the API Jungle âĹŮ. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 48–61, New York, NY, USA, 2005. ACM.

[11] D. McAllester and D. Rosenblitt. Systematic Nonlinear Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 634–639. AAAI, 1991.

[12] M. Shaw. Procedure calls are the assembly language of software interconnection. In *Proceedings of the Workshop on Studies of Software Design*. Springer, 1993.

[13] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[14] Sun Microsystems. Javabeans api specification. http://java.sun.com/products/javabeans.

[15] The OSGi Alliance. Osgi service platform release 4 version 4.2 core specification. http://www.osgi.org/Download/Release4V42.

[16] R. Vasa, M. Lumpe, and J. Schneider. Patterns of component evolution. In *Lecture Notes in Computer Science*, volume 4829, pages 235–251. Springer-Verlag, 2007.

[17] A. Zaremski and J. Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 1997.