Promises: Limited Specifications for Analysis and Manipulation.

Conference Paper *in* Proceedings - International Conference on Software Engineering · January 1998

DOI: 10.1109/ICSE.1998.671113 · Source: DBLP

3 authors:

Edwin C. Chan
Carnegie Mellon University
3 PUBLICATIONS **608** CITATIONS
SEE PROFILE

John Tang Boyland
University of Wisconsin - Milwaukee
74 PUBLICATIONS **1,449** CITATIONS
SEE PROFILE

William L. Scherlis
Carnegie Mellon University
80 PUBLICATIONS **4,940** CITATIONS
SEE PROFILE

Promises: Limited Specifications for Analysis and Manipulation*

Edwin C. Chan
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 3076
chance@cs.cmu.edu

John T. Boyland
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 3738
John.Boyland@acm.org

William L. Scherlis
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA USA 15213
+1 412 268 8741
scherlis@cs.cmu.edu

ABSTRACT

Structural change in a large system is hindered when information is missing about portions of the system, as is often the case in a distributed development process. An annotation mechanism called *promises* is described for expressing properties that can enable many kinds of structural change in systems. Promises act as surrogates for an actual component, and thus are analogous to "header" files, but with more specific semantic information. Unlike formal specifications, however, promises are designed to be easily extracted from systems and managed by programmers using automatic analysis tools. Promises are described for effects, unique references, and use properties. By using promises, a component developer can offer additional opportunity for change (flexibility) to clients, but at a potential cost in flexibility for the component itself. This suggests the possibility of using promises as a means to allocate flexibility among the components of a system.

KEYWORDS

Aliasing, effects, program restructuring, headers, implementation flexibility, limited references, effect regions, separate compilation, unique references

*This material is based upon work supported by the National Science Foundation under Grant No. CCR-9504339 and by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, the National Science Foundation, or the U.S. Government.

20th International Conference on Software Engineering (ICSE '98), April 19–25, Kyoto, Japan.

0-8186-8368-6/98 $10.00 © 1998 IEEE

1 INTRODUCTION

Analysis and alteration of a part of a software system is hindered when information is missing about the remaining portions of the system and conservative assumptions must be made. We describe a language of limited specifications, called *promises*, to express properties that are significant in making structural change in systems, but that also can be easily extracted from program components with automatic tools playing a primary role.

Promises offer a compromise between the minimal type information managed in "header" files and the more extensive semantic information that might be in an architectural specification. The additional semantic information provided by the promise annotations can yield more precise analyses, and hence enable a broader range of program manipulation and restructuring to be carried out with a guarantee of soundness. Structural modifications are common in software evolution; examples include restructuring a class hierarchy, copying an encapsulation for tailoring to a specialized subset of uses, changing data representation, and abstracting a new function definition from existing code.

Promises are intended to be sufficiently easy to specify and validate that they can be used in routine design and programming activity, particularly program evolution using tool assistance. In this regard, promises are similar to the C-language annotations of tools such as LCLint [7]. The LCLint annotations focus on a different set of goals—specification of encapsulations and error-detection. For example, LCLint provides an annotation language for the explicit specification of programmer-intended abstract data type boundaries, which are not directly expressible in the C language. Annotations such as these support analysis algorithms that perform error detection such as ensuring that the encapsulations are respected.

In this paper we define portions of a language of promises for Java. The promises relate to effects, aliasing, and use properties, and can be attached to a variety of different program elements. The promises enable a component implementor to offer "contracts" to clients that permit clients to make structural changes that might not be sound without the guarantee provided by the promise. For example, guarantees concerning the effects of a method may enable safe reordering of computation. A component implementor may require promises of the clients as well—we call these requirements *demands*. Our thesis is that there is a relatively small set of promise types needed to enable a broad range of restructuring manipulations to be carried out safely, that the introduction of promises into code can be largely automated, and that promises can be effectively managed and changed by programmers and tools. In general, however, promises cannot feasibly be checked at runtime. This regime of promises is being incorporated into an experimental tool that supports systematic source-level structural change in Java programs. This will enable us to explore the practicality of this approach for realistic case studies.

A key design issue is how to formulate promises that are useful to component clients but do not unduly restrict choices in component implementation. For example, promises concerning effects indicate what portions of mutable state can be read or modified. Specifying effects directly in terms of fields (specific variables in classes) may unnecessarily expose implementation details and restrict subclasses. Instead, we employ the concept of abstract regions of an object, onto which the fields are mapped. This enables useful local analysis while retaining implementation flexibility.

In the Section 2 below, we introduce promises relating to effects, aliases, and structure. While promises about a component grant additional flexibility to its clients, they also limit flexibility for evolution of the component itself. This suggests that, with appropriate tools, a (mature) regime of promises could serve as a "currency of flexibility" in the management of larger software systems, with use (or non-use) of promises providing a mechanism to "allocate" engineering flexibility among the elements of a system in response to extrinsic engineering factors. In Section 3 we consider some technical issues concerning the feasibility of this prospect, particularly the role of tools in managing promises.

2 Promises

Promises are supra-linguistic formal annotations to programs. That is, they have precise meaning, but are not part of the actual programming language definition. They are meant to be interpreted by program analysis and manipulation tools and (generally) ignored by compilers. Annotations can be applied to several different kinds of abstract-syntactic types. They can be made syntactically concrete as formally structured comments (as in Anna, Javadoc, and LCLint) or by direct integration into the syntax (and then presumably removed by a preprocessor to the compiler). For convenience in presentation, we use the latter approach in this paper. (Regardless, tools can be used to display promises to a programmer using any convenient syntactic form, even eliding promises not of current interest.)

Semantically, promises about a component restrict choices for its implementation and evolution (once clients rely on these promises). But in client code, these restrictions have the opposite effect—they enable more precise analysis, and hence they enable a broader range of meaning-preserving changes in client code.

In many software engineering processes, the kinds of information expressed in a promise may normally be part of design documents or otherwise be expressed in organizational memory. There are obvious advantages to capturing a promise explicitly: The representation is formal and precise, and it is in simple programmer-oriented language. Tools can interpret the promise and provide analysis support, for example to validate hypotheses programmers may have about a system, or to suggest new promises that could be offered. And tools can exploit promise information to help a software engineer carry out larger-scale structural change not otherwise feasible particularly on incomplete programs [10, 17, 19].

As with other kinds of specifications, however, there are costs—each promise must be identified and expressed. Nonetheless, we suggest that promises differ from full functional specifications in two important respects. First, the limited semantic content of promises makes them easier to state, validate, and exploit. Second, like LCLint, promises offer an incremental approach to adoption by software engineers—identifying and expressing an individual promise is a very small increment of effort, that, in general, should yield some increment of value in client flexibility, analysis results, and manipulations enabled. Promises enable software engineers to be selective about both the specific elements of a system that they annotate and the extent of promises made for those elements.

Our selection of particular semantic properties to express as promises is guided by the analyses needed to enable a wide range of evolutionary changes in programs, particularly changes to shared abstractions in larger software systems. "Code rot" can be considered to be the persistence of abstractions beyond their time. Change to abstractions can be facilitated when they are better encapsulated, when more of their properties are directly expressed, and when their uses are more precisely understood. For this reason, promises are *not* designed to address functional properties, but rather the more forgettable "bureaucratic" ones—semantic properties such as effects and mutability, and structural properties such as where the call sites of a method are located.

Feasibility of the overall approach requires that promises be easily expressed and understood by programmers, and that, with use of appropriate tools, promises be easy to offer (extract directly from code) and validate (when explicitly claimed). The three classes of promises considered in this paper can, in general, be extracted from code using automatic tools with a limited amount of programmer guidance. Validation of promises claimed for a component (or explicitly sought by a client) is also straightforward. Validation is done by analysis; runtime checks are not practical for most of the promises we consider (see Figure 1).

An example of a simple restructuring manipulation frequently carried out by a programmer is the abstraction of some existing fragment(s) of code into a new function definition (e.g. a method declaration in Java). This might be done simply to reuse those code fragments elsewhere in the program, or done as part of a more complex manipulation, such as redistributing code while changing the internal representation of a class. Carrying out the abstraction requires identifying the code site of the existing code to abstract as the body of the new method definition, the sites of contained code fragments that will remain at the calling site as actual parameters, a specification of the order in which the parameters will appear, a site for the new definition, and names for the new method and its formal parameters. Assurance of soundness of this seemingly simple manipulation requires several analyses that are supported by semantic and structural promises. For instance, both order and frequency of computation may be affected by abstracting some computation into an actual parameter. Effects promises, as described in Section 2.1 and made more precise in Section 2.2, allow the determination of safe reorderings. Name bindings may be affected by the relocation of code to a new site (e.g., across an encapsulation or scope boundary) and by the introduction of new formal parameter names and a new name for the abstracted method. Section 2.3 describes a way to identify the uses of the class whose interface is being changed by the introduction of this new method. Together, these three kinds of promises can be used to guarantee soundness of an application of this manipulation, as well as others.

2.1 Effects analysis

The abstraction of a code fragment into a new method definition is one of many cases where reordering of code is required in order to enable a more significant structural change. The analysis primarily involves identifying effects and dependencies on them. Effects, for our purposes, include reading and writing of modifiable store, since modeling of data dependencies requires us to track "read effects" as well as write effects.

`reads`, `modifies`— Possible effects of methods.

`unique`, `unshared`— Unaliased references.

`limited`— Reference not to be stored.

`immutable`(*)— Referenced object is never modified.

`instanceof`(*)— Referenced object is an instance of one of specified concrete classes.

`usedBy`— Clients of a field, method, or class.

Figure 1: Some promises and demands (* = not discussed in this paper).

Consider the following class that encapsulates a variable:

```
class Var
{ private int val = 0;
  public void set(int x) { val = x; }
  public int get() { return val; }    }
```

Suppose, in our method abstraction scenario, we have a block that contains an expression `v.get()`, which may in a conditional or a loop, and we wish to abstract the block to create a new method `M`, but with the expression `v.get()` remaining behind as an actual parameter in the call that replaces the block. Let `v` be non-null (i.e., it points to an allocated object). Then it appears that as long as nothing else in the block modifies the private instance variable in `v` (for example if we know `v` is not aliased anywhere), it would be safe to extract the expression. This information is not sufficient, however: `v` may in fact be an instance of a *subclass* of `Var`, perhaps of a class such as the following:

```
class NoisyVar extends Var
{ public int get()
  { System.out.println("Var is read!");
    return super.get();     } }
```

This `get` method has an effect. If `v.get()` were in a conditional or inside a loop, then computing it in advance as a parameter could lead to extra output or missing output. We could designate `Var` as a final class (and this designation is a kind of promise). But if subclasses to `Var` are to be allowed, we need to be able to offer a commitment to clients of `Var` that its subclasses don't have any more effects than `Var` has.

In this example, we concretely express the promises by adding extra modifiers to method headers (see prior discussion concerning syntax). We annotate `Var` with `reads` and `modifies` promises in the style of Java `throws` declarations:

```
class Var
{ private int val = 0;
  public void set(int x) modifies val
    { val = x; }
  public int get() reads val
    { return val; }    }
```

The `reads` promise denotes that the method can access `val` but have no other effects. The `modifies` promise indicates that `set` may read and write `val`, but no other variables. As with `throws`, we require these promises to be respected by overriding methods as well. If the class `NoisyVar` were added to the system, the tool would detect the lack of conformance in the overriding `get` method, and alert the developer. The `reads` and `modifies` promises thus allow clients of `Var` to reason about the effects of method calls without needing to know the bodies of the methods, as would be the case in a distributed development process.

Unfortunately, there are two problems with this approach:

- The names of private instance variables are revealed in the headers of methods, violating the encapsulation.
- Subclasses may be unnecessarily restricted.

For an example of the second problem consider the following class:

```
class UndoableVar extends Var
{ private int saved = 0;
  public void set(int x) modifies val, saved
    { saved = get();
      super.set(x); }
  public void undo() modifies val, saved
    { set(saved); }    }
```

Here `set` does not obey the restriction of the method that it overrides and thus would not be legal. Nonetheless, the additional effects do not affect whether an expression such as `v.get()` can be reordered with respect to an expression such as `w.set(42)`.

Thus instead of listing instance variables, effects promises specify *regions* that are read or modified. Regions are abstractions for sets of fields. A class may have multiple regions each of which refers to a disjoint set of instance variables. Figure 2 shows how `Var` and `UndoableVar` are annotated with regions. Since they are abstract, regions permit information hiding and offer control over implementation flexibility. Because they are disjoint, regions permit effects analysis to determine safe reorderings.

```
class Var
{ region Value;
  private int val in Value = 0;
  public void set(int x) modifies Value
    { val = x; }
  public int get() reads Value
    { return val; }    }

class UndoableVar extends Var
{ private int saved in Value = 0;
  public void set(int x) modifies Value
    { saved = get();
      super.set(x); }
  public void undo() modifies Value
    { set(saved);  }    }
```

Figure 2: An example of effects promises

In Java, fields may be declared "static," that is, shared by all objects of a class. Similarly, regions may be declared "static," in which case the determination of effect overlap takes the sharing into account.

The reads and modifies promises describe the effects of method calls. A method may have effects on objects other than the receiver; it may read or write objects passed as parameters, objects available in public static fields (for example `System.out`), or objects indirectly reached through one of these other objects. In the reads and modifies promises, p`.`r refers to region r of the object passed as parameter p, C`.`s`.`r refers to region r of the object in public static field C`.`s, and C`.`r refers to region r of *any* object (or set of objects) of class C. The latter notation involves a loss of precision. The shorthand "`*`" refers to all regions of the object. Analogously, p`.*`, C`.`s`.*` and C`.*` refer to all regions of (respectively) an object passed as a parameter, an object in a public static field, and any object of class C. Finally, the region `*.*` (with the same meaning as `Object.*`) refers to all regions of all objects. If a method does not declare any effects (and analysis has not been done to infer effects promises) then the most conservative approximation is assumed, which is that it modifies `*.*`.

Effects annotations can be verified using intraprocedural analysis (that is, analysis local to a method body). The process is very similar to that used by compilers to check `throws` clauses in Java. One first determines the local effects (reads and writes to fields) and then adds the effects of every method call in the body using the annotations on the statically selected methods. The resulting set of effects is then compared with the effects declared for the method. Effects can be ignored if they occur on newly created objects. More generally, the existence of "unique" (unaliased) references (as explained shortly) can make effects inference more precise.

Furthermore, once a developer has specified the regions for each field, an *inter*-procedural analysis can be used to infer the effects of a large body of code all at once and get a system of promises in place. The technique outlined above computes the effects of each method in terms of the effects of other methods. A set of constraints can thus be induced from the method bodies, and the least fixed point of these constraints will give a valid set of promises. Some of the resulting effects promises may prove too strict for subsequent development, and may need to be liberalized.

2.2 Unique References

As shown above, effect specifications are useful in assuring soundness of manipulations that require reordering code. If two effects take place in distinct regions (or in regions of distinct objects), they cannot conflict. Observe that if the effects take place in a region of an object that has just been allocated, they are invisible to the caller and need not be declared in an effects promise. This is because the "just allocated" objects are *unique references*—unaliased references that cannot be accessed in any other context. In this section, we formalize and generalize this concept into a separate kind of promise.

A promise that a reference is unique also ensures that the object to which it points can be copied without disturbing the meaning of the program (because object identity is irrelevant). This scenario may occur as part of a manipulation that copies a class to make two specialized classes. At some point, one may need to treat an object of one derived class as one of the other class. If the reference to the object is unique, the contents of the object can be safely copied into a new object of the other class and the original object abandoned.

Suppose we wanted to swap the order of the two calls to `Pair.copy()` in the following example:

```
class Pair
{ Object a, b;
  static void copy(Pair dst, Pair src)
    reads src.* modifies dst.* { ... } }
class Main
{ static void main(Pair a, Pair b, Pair c)
    { copy(b, a);
      copy(new Pair(), c); }  }
```

Looking at its region specification, we may know that the first call, `copy(b, a)`, could modify `b`. In the absence of other information, a conservative analysis forces us to assume the worst, which is that it could also change `a`, or any other object of class `Pair` (or its subclasses), because any of those could be aliases of `b`. Similarly, the same call could also read `a` and any of its aliases. In other words, we could not safely reorder the two statements.

If all references are known to have no aliases, however, there is no confounding, we can achieve maximal precision, and the reordering change is sound. One obvious way to achieve that situation is to never allow any aliases, but, in general, that restriction is impractical (but see Baker's Linear Lisp [3]). Still, these insights motivate the concepts behind *unique* parameters and return values. References (other than `null`) passed as such parameters or returned from methods or constructors so declared can thus be counted on to be distinct from any reference stored in another variable or returned by any method. This is an aggressive approach to alias control that facilitates many structural manipulations, particularly those that involve introduction and elimination of copying, and the introduction of destructive operations on data structures.

A reference is *unique* when it is initially created, and also when it is the last live reference to a unique object passed as a parameter or returned from a method or constructor. (Note that while a unique reference is unaliased, references in fields of the referenced object might not themselves be unique.) Observe that a unique annotation on a *parameter* is in fact a *demand*—a promise required of each client.

The return value of a method is declared unique in the following manner:

```
unique Var makeUniqueVar()
{  Var v = new Var();
   ...
   return v;  }
```

This method creates a unique reference, performs some other tasks and then returns it. As long as the other tasks do not create lasting aliases, uniqueness is preserved. There is a potential problem with `makeUniqueVar`: the value returned by `new Var()` need not be unique, since the reference could be compromised by any of the various constructors that were called in the process. Looking at the definition of the class `Var`, we note that its constructor does not create an alias to the reference (`this`) it gets from the constructor for `java.lang.Object`. Thus, we should annotate it with a promise that the reference returned will be unique.

We do not annotate *local variables* with 'unique' promises, because such annotations would seem to preclude harmless local aliases, and, more seriously, because a local variable may only be unique during part of its lifetime. Thus, at each execution point, the tool keeps a record of which locals contain references that have no external aliases and for each of these 'externally unique' locals, which other locals may alias it.

Passing a unique reference as an actual parameter (including implicitly using an instance method of the unique reference) may involve the loss of uniqueness since the called method may introduce lasting aliases. (In Java, objects are not copied when they are passed as parameters; method calls induce the creation of aliases.)

We can, however, take advantage of the fact that the scope changes on transfer of control from caller to callee, and weaken our notion of 'aliasing' by making it relative to what is in scope and accessible to a method. This insight suggests we can allow unique references as parameters as long as the method promises that it will not create lasting aliases of the reference. This restriction is accomplished by designating the formal parameter as a *limited* reference parameter. (Observe that when a method's formal parameter is `limited`, this is a promise, not a demand.) Limited references may not be assigned to a field either directly or indirectly through another method, and cannot be returned from methods. The first constraint is to keep from creating globally accessible (and potentially long-lasting) aliases, while the second is to keep from introducing 'unknown' aliases in calling frames. Declaring a *method* to be limited has the effect of declaring the receiver `this` to be limited. If the receivers for `get` and `set` are limited, then we can declare a method such as the following:

```
public static void bump(limited Var v)
    modifies v.Value
{ v.set(v.get()+1); }
```

Since the parameter is limited, this method can be called with a unique reference without creating an alias, and the unique reference will remain unique on return from the method. Thus `bump` could be called in the body of `makeVar`.

When we pass a unique reference to a method as limited parameter, the effects analysis of the called method cannot use the fact that the reference is unaliased. But since the caller's copy of the reference cannot be accessed during the dynamic context of the call, one can pass a unique reference as a *limited unique* reference instead (as long as the reference isn't passed as more than that one parameter). Limited unique parameters are unaliased within a certain dynamic context and have the properties of both unique and limited parameters. As with unique references, limited unique references may be passed to methods as either limited or limited unique parameters.

Up until this point, we have only considered parameters and return values as unique. Although useful in their own right, they can only store unique references temporarily until they go out of scope 'forever', and so we need more permanent places for them, which are *unshared fields*. Unshared fields are really no more than a form of unique variable, except allocated in the store as part of an object. Similarly *static unshared* fields hold unique references associated with the class object.

A problem with unshared fields is that they invalidate the assumption that a variable, and thus the reference it contains, is only accessible from a single frame. For instance, suppose that within a class with an unshared field, we call a method with the reference from this field as a parameter. Since there is no guarantee that the object with the unshared field is itself unique, we cannot assume the field will be inaccessible during the call.

A simple solution to this is to use a promise that guarantees that the code being called will not access that field. Thus, we only allow an unshared field to be an actual parameter if it is passed as limited and the method to which it is passed does not read or modify the region for that field (that is, the method must have the appropriate reads and modifies promises). As a consequence, if we have existing code that does need to access that region, we may have to nullify the unshared field during the call.

A tool can infer `limited` promises for method parameters, and `unique` promises for the return value: the tool starts with tentative `limited` promises for each parameter and local variable, a `unique` promise for the return value. Then the tool checks the promises, removes any that cannot be verified, repeating until the method body checks. When the verification finally concludes successfully (at worst when all the tentative promises has been removed), any remaining promises are verifiable, by definition.

The uniqueness of parameters must be inferred in a different way, because rather than limiting the implementation of the method, uniqueness of the parameters limits the callers (i.e., they are demands). If all the call sites of the method are known (using the `usedBy` promise explained in the following section), the actual parameters can be examined for whether they are unique references (or perhaps limited unique references) and if all are, the formal parameter can be given the appropriate annotation.

Inference of `limited` and `unique` promises can be performed on a group of methods after the developer has identified unshared fields. First the tool tentatively assumes the most restrictive scenario: all parameters are limited, all return values unique, all local variables limited, and finally a `unique` promise for parameters of methods for which all call sites are known. All methods are checked and any unverifiable tentative promises are discarded, iterating until all promises can be verified. If effects promises are also to be inferred, they should be inferred once the uniqueness promises are in place, because uniqueness makes effects inference more precise.

This inference method can reduce the work of adding promises, but may fail to come up with useful annotations when the code deviates slightly from the rules required of limited or unique values. In this case, the developer may first make a few promises in key places, then perform the inference, examine where the hand-added promises fail to verify, and iterate.

2.3 Structure

Changes to the interface or functionality of a component is likely to require concomitant changes to client code. In a small scale development environment where a single person or a small team controls all the source code, all the affected code can be identified. When the system is made up of separate subsystems that are built by different teams or obtained as binaries from a third-party the situation becomes problematic. It may be difficult for a developer to be aware even of the extent of usage of a component, much less its nature. This lack of information may make changes to a class impossible without risking the soundness of the system.

Making a complete-program (i.e., closed-world) assumption implies that all use-sites are contained in the code provided. Many analysis techniques require this assumption, and this limits their value. Promises offer a mechanism of surrogacy that enables analyses on incomplete systems. (In this regard there is some analogy with the internal annotation that occurs in some complete-program inter-procedural analyses.)

Our approach to this brittleness problem has two elements. First, the sort of semantic promises explored in the previous two sections can enable soundness to be assured for many structural and semantic changes that would otherwise be too risky in the absence of the complete program text. Second, we introduce an additional form of promise that helps define the scope of usage of a component more precisely than through the usual visibility attributes (`private`, `protected`, and so on) in the programming language. By explicitly annotating declarations with information about their use-sites, we can in many cases regain this sense of closure by bounding the amount of code that may require analysis and modification.

For Java, this means that we annotate classes, fields, and methods with enumerations of all the classes (other than the owning class) that could contain a use-site. Such promises need only be added if and when they are needed in order to enable manipulations. Any declaration can be annotated with a "used by" list of the following form: $\mathit{UsesList} := \texttt{usedBy}\ (\mathit{ClassName})*$

Directly implemented, this scheme would likely be overly cumbersome, since both the number and size of annotations will increase significantly with the size of a system. In addition, small changes to code may necessitate large-scale changes in promises throughout the system.

Annotations would need to be updated each time a new client class began to use a component class. But annotations would not need to be changed when further use-sites are added within an existing client, nor would they need to change if any of those classes stopped accessing those fields (though the resulting promises would then be more conservative than necessary and a tool might request source access to the superfluous classes).

A two-fold refinement addresses these difficulties: we use abstract sets of client classes, and also use Java's package and inheritance structure to implicitly limit uses. First, rather than repeatedly specifying a set of classes, the set can be given a name:

$$\begin{aligned} \mathit{SetDecl} &:= \texttt{usedSet}\ \mathit{SetName} = \mathit{UsedSet}\ ; \\ \mathit{UsedSet} &:= (\mathit{ClassName} | \mathit{SetName})* \\ \mathit{UsesList} &:= \texttt{usedBy}\ \mathit{UsedSet} \end{aligned}$$

A set name thus denotes the union of the explicitly-mentioned classes and sets.

By employing used sets, the task of writing promises can be factored, and this task can be supported by tools, with developer involvement in managing the tradeoff between the cost of the annotation process and the precision obtained.

Observe that Java already has named sets, called packages, which are implicitly created by identifying a class as an element, but which are never explicitly enumerated.

For example, if the class `fluid.util.AssocList` specifies the following set definitions (using simple class names where possible):

```
usedSet fluid.util =
          AssocList, AssocKeyEnum,
          AssocValueEnum, Stack;
usedSet fluid.util.AssocList* =
          fluid.assoc.FixedLenAssocList,
          fluid.assoc.MonotonicAssocList;
```

The system can then use them to complete the following implicit annotations for the various access level modifiers:

```
private:   usedBy AssocList
default:   usedBy fluid.util
protected: usedBy fluid.util,
                  fluid.util.AssocList*
public:    usedBy *
```

Here the `*` extension of a class name refers to the set of all its subclasses (not just its direct subclasses) and the `*` in the `usedBy` clause refers to the unbounded set of all classes. Without the explicit used set definitions, the implicit annotations are equivalent to `usedBy *`.

The `usedBy` annotations can be checked whenever a piece of client code refers to an entity in a different class, with an error indicated when the class of the client code is not in the used set. It is trivial for a tool to infer used sets if the whole program is available. Otherwise, the developer can make a large-scale promise (such as identifying all the classes that might use some class) which then could be used as part of an ongoing iterative process.

2.4 Summary

In this section, we have described a number of promises: effects promises enable safe reordering of code; unique promises enable safe insertion of destructive operations and of copying, and they make effects analysis more precise; "used by" promises allow escape from complete-program assumptions by limiting the scope of possible places in the code where a declaration can be used. Use of these promises can enable soundness guarantees for many kinds of program-restructuring activities that would otherwise be unsafe to carry out in larger systems and in distributed development efforts.

We suggest that promises such as these, unlike more complete functional specifications, can be directly managed by programmers working with analysis and manipulation tools (employing static analysis algorithms rather than general-purpose theorem proving), with tools taking a major role in identifying and validating promises, and tracking their use in manipulation. In those cases where a tool fails to validate a promise or assure a demand (due, for example, to lack of availability of a particular component), a dependency tracking mechanism can allow a software engineer can take responsibility for the promise, and resolve its status (and the status of changes which depend on it) at a later time.

3 MANAGING PROMISES

Promises offer a "currency of flexibility" for developers and maintainers of software, in the sense that the addition and removal of promises regulates the balance of flexibility between a component and its clients. When a component offers many promises, clients gain flexibility in the kinds of restructuring changes they can make. But an overly comprehensive set of promises can constrain the evolution of the component itself by limiting options for change within the component.

Thus, in general, the management of promises involves decisions to add and remove promises (and demands), and techniques to facilitate the making and implementation of those decisions.

The `throws` clause in Java serves as a kind of promise, and its design is illustrative of more general issues of promise management. In Java, a method must declare the classes of exceptions that may be thrown during its execution. This set of thrown exceptions is computed from the types of exceptions used in `throw` statements and declared as thrown by methods called from that method. From this set are removed the exceptions caught by `catch` clauses. The remaining (uncaught) exceptions should be declared in the header of the method. This determination of which exceptions could be thrown out of a method body is necessarily conservative. For instance, suppose a method is declared as possibly throwing some exception, but it is only called under conditions in which no exception will be thrown. To a Java compiler, it will be appear the potential exception is uncaught and should be declared in the `throws` clause of the calling method. Therefore, in order to keep the restriction from being too burdensome, those exceptions from the class `RuntimeException` and its subclasses need *not* be declared. This class covers common runtime errors such as dereferencing a null pointer or indexing an array outside of its bounds. When a developer defines an exception class, it can be made a subclass of `RuntimeException` to avoid the need to declare it in `throws` clauses, but then the benefit of the throws clause is lost. In order to avoid such dilemmas with promises, the developer can take formal responsibility (recorded by a tool) for promises that cannot yet be verified.

3.1 Offering promises

A naive analysis suggests that there are two possible approaches that can be followed in determining the correct set of promises to be offered at any point in the development or evolution of a system. These are building up and building down. The "building up" approach suggests that developers add promises, either reactively when the safety of a change to client code needs a particular promise to be made, or preemptively when there is high value in client flexibility at low cost in component flexibility. In this case, in order to offer the additional promise, the developer may need to extract promises regarding other components. Of course, the promises may not be immediately forthcoming, in which case the developer could choose to leave the promise unverified, and so record (using a dependency management tool) a potentially inconsistent dependency for later resolution.

The "building down" approach, on the other hand, suggests that developers use automatic tools to identify a large set of valid promises that can be offered relating to software component. This is possible for all three kinds of promises described in Section 2, with programmer interaction needed only in specific instances, for example to define regions and identify unshared fields. In this case, the programmer may need to serve as a moderating force to assure that the balance of flexibility does not swing too far from component developer to component client as a result of an excessive weight of automatically generated promises. This is accomplished in two ways: prospectively by monitoring and managing the promises that are to be published for a component, and retrospectively by using tools to manage dependency information that identifies which promises actually contribute to assuring the soundness of code changes in client code. This dependency information can assist programmers in removing or weakening promises that have no dependencies.

In practice, extrinsic engineering considerations are likely to be the primary guide for decisions relating to this balance of flexibility. Such considerations would indicate, for example, which components are most likely to evolve over an anticipated life-cycle or across a product-line. The benefit of promises is that they offer a more precise and explicit way to manage this balance of flexibility, while assuring soundness of structural changes where they are most likely to be made. It remains an empirical issue to see what are the costs—how the balance can be effectively managed in practice, what are appropriate roles for tools, and how the implicit negotiation between component and clients can be supported.

3.2 Breaking promises

There are two ways in which a promise may be broken. It may turn out that a recorded but unvalidated promise cannot be kept. It may also happen that subsequent modifications to the code break the promise or violate a demand. In either case, it is important to find out where the promises have been used so that the extent of the problem can be determined.

4 RELATED WORK

Whole program analyses that identify properties such as aliasing and side-effects are too numerous to mention. Rather than helping the programmer to constrain the behavior of separately developed components (as in the case with promises), these analyses are usually used by sophisticated optimizing compilers. Sometimes, the compilers are assisted with promise-like annotations that make up for a lack of precision in the analysis by locally overriding the analysis results. Unlike promises, however, these annotations are not interprocedural in nature and they are not supported by a system that permits verification options ranging from fully proved to simply trusted.

Flanagan and Felleisen [8] describe a set-based dataflow analysis that integrates the results of separately analyzing components. As an aside, they propose summarizing the voluminous machine-generated set constraints with signatures that approximate the exact constraints. They speculate that these signatures could be programmer-specified.

Other researchers have used the technique of enhanced "header files" for breaking intractable global consistency issues into locally manageable statically checkable pieces. As noted above, LCLint [21, 6] used annotations on pointer types in C in order to statically check for dangling pointers and storage leakage. This project was carried out in the context of Larch [11], a formal specification and theorem proving system, which similarly used a variety of proof-supported resources for partially or fully specifying the behavior of separately developed entities. In particular, Larch supports mutability annotations. 'Extended static checking' [5, 15], which also uses a theorem prover, has been incorporated into experimental Modula-3 compilers. In particular, lock level annotations were successfully used in the Trestle window system to ensure the absence of deadlocks.

Jackson's Aspect system [13] uses a similar abstraction mechanism to our regions in order to specify the effects of routines on abstract data types. He uses specifications for each procedure in order to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects, in order to check for missing functionality; whereas in our system the specified effects must include all effects of a method.

Effects were first studied in the FX system [9], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Taplin [14, 20]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successful for the former purpose in later work [22, 1]. These researchers also make use of a concept of disjoint "regions" of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Reynolds [18] originated the idea of "syntactic control of interference." The intent was to develop an Algol-like programming language that used syntactic properties to eliminate "anonymous channels of interference." Region promises are similar in intent, but differ in realization. They are selectively applied (to parts of a Java program) and allow specification of effects on parts of an object.

Adding notation for unique pointers was studied by

Minsky [16] in the context of Eiffel. Baker's 'use once' variables [4] show the fundamental rules that any such system should follow. Hogg [12] presented a notation for Smalltalk that enabled the identification of "islands," regions of objects accessible only through a single bridge point. This work has been made simpler and more powerful by Almeida [2], where it is shown that a range of types: balloon types, opaque balloon types, and value types can be declared and checked in the context of an Algol-like language. Objects within an island or balloon differ from our unshared fields in an object because aliases are permitted within an island. Furthermore, we permit an object to have both shared and unshared fields.

Our work differs from such language extensions because we do not rely on a single set of rules to verify promises. Promises additionally may be verified by external tools, or even by hand. This flexibility permits promises to be useful as soon as they are deployed rather than requiring an entire program to be annotated before it can be accepted.

Griswold and Notkin [10] have a tool to perform meaning-preserving restructuring on Scheme programs and describe a number of basic transformation steps. They keep the source, a program dependence graph (PDG) and control-flow graph (CFG) up-to-date with parallel transformations. The transformations are shown to preserve meaning using whole-program analysis. Their techniques would need to be modified to handle large or incomplete programs.

REFERENCES

[1] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *ACM SIGPLAN PLDI'95 Conference*, pages 174–185, New York, June 1995. ACM Press.

[2] P. S. Almeida. Balloon types: Controlling sharing of state in data types. In M. Akşit and S. Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming (11th European Conference)*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59, Berlin, June 1997. Springer-Verlag.

[3] H. G. Baker. Lively linear Lisp — 'Look Ma, no garbage!'. *ACM SIGPLAN Notices*, 27(9):89–98, Aug. 1992.

[4] H. G. Baker. 'Use-once' variables and linear objects: Storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, Jan. 1995.

[5] D. L. Detlefs. An overview of the extended static checking system. In *First workshop on Formal Methods in Software Practice*, 1996.

[6] D. Evans. Static detection of dynamic memory errors. In *ACM SIGPLAN PLDI'96 Conference*, pages 44–53, New York, May 1996. ACM Press.

[7] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 87–96, New Orleans, Louisiana, December 6–9 1994.

[8] C. Flanagan and M. Felleisen. Componential set-based analysis. In *ACM SIGPLAN PLDI'97 Conference*, pages 235–248, Las Vegas, Nevada, June 15–18, 1997.

[9] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, MIT Laboratory for Computer Science, Sept. 1987.

[10] W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM TOSEM*, 2(3), 1993.

[11] J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[12] J. Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91*, pages 271–285, New York, Nov. 1991. ACM Press.

[13] D. Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, Apr. 1995.

[14] P. Jouvelot and D. K. Gifford. Algebraic reconstruction on types and effects. In *18th ACM POPL Conference*, pages 303–310. ACM Press, New York, Jan. 1991.

[15] K. R. M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, Pasadena, California, USA, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[16] N. Minsky. Towards alias-free pointers. In P. Cointe, editor, *ECOOP '96*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Berlin, Heidelberg, New York, July 1996. Springer-Verlag.

[17] W. Opdyke. *Refactoring Object-Oriented Frameworks.* PhD thesis, University of Illinois, 1992.

[18] J. C. Reynolds. Syntactic control of interference. In *5th Annual ACM POPL Symposium*, pages 39–46, January 23–25 1978.

[19] W. L. Scherlis. Small-scale structural reengineering of software. In *ACM SIGSOFT 2nd International Workshop on Software Architecture*, pages 116–120. ACM Press, Oct. 1996.

[20] J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.

[21] Y. M. Tan. Formal specification techniques for promoting software modularity, enhancing software documentation, and testing specifications. Technical Report MIT/LCS/TR-619, MIT, June 1994.

[22] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *21st ACM POPL Conference*, pages 188–201, New York, Jan. 1994. ACM Press.