Automatic Protocol-Conformance Recommendations

Ernesto J. Alfonso
Carnegie Mellon University
ealfonso@andrew.cmu.edu

Abstract

Misuse of reusable components in software is common. Systems of software analysis based on formal specifications provide a mechanism for automatically detecting non-conformance to protocols. The focus of this research is to automatically generate task-specific user recommendations for correcting misuse of arbitrary protocols using results from software analysis systems.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

General Terms Verification

Keywords protocol, automatic fix, suggestions, error message generation, predicate logic, specifications

1. Introduction

As the utility of a reusable software artifact increases, so does the complexity of its public interface and usage protocols. This often leads to errors of misuse at the points of contact of reusable artifacts with their users. In order to explicitly enforce proper usage protocols, formal specifications of reusable components may be used by software analysis systems in order to automatically verify conformance to these protocols. However, upon discovery of a misuse, any user feedback provided directly by such systems must be in terms of the protocol specification, which is unusable to the user who is not already an expert in the protocol. The contribution of this research is a system which automatically generates task-specific, user level recommendations for correcting misuse of protocols, based on results from analysis mechanisms with formal specifications.

2. Motivating Example

Consider the following example of a protocol misuse,

Listing 1. Misuse of the iterator protocol

```
1
2  public Object clumsyPoll(Queue queue) {
3    Iterator iter = queue.iter();
4    Object o = null;
5    if (iter.hasNext())
6      o = iter.next();
7      iter.remove();
8    return o;
9  }
```

Listing 2. Current feedback from Fusion

```
1  Broken constraint: Removable(iter)
```

Currently, the formally specified iterator protocol along with the Fusion[1] program analysis provide the feedback in listing 2 in response to the protocol misuse in listing 1 (i.e. Iterator.next must be called before Iterator.remove). Despite listing 2 being a relatively intuitive error message, which might serve a user already familiar with the very simple protocol, this type of error still refers to the internal representation of Fusion and the Iterator specification implementation, and in general does not provide any direction towards fixing the problem. Note also, whereas the example in listing 1 is intentionally simple, real life examples of misuse tend to be more involved.

Our goal is to provide task-specific recommendations which the user can understand, such as the following suggestions independently addressing the misuse in listing 1:

- Move call to "iter.remove()" to line 7 within if-block starting at line 5.
- Remove the method call to "iter.remove()" on line 7
- Call "iter.next()" before line 7

Approach

Consider,

- An analysis system which relies on a formal specification language (e.g. Fusion and its spec. language)
- A reusable component which usage protocol is specified for the above system (e.g. Iterator)
- A user program U which uses the reusable component improperly, breaking specification C (e.g. U, C are listings 1, 2 resp.)

SPLASH'11 Companion, October 22–27, 2011, Portland, Oregon, USA.
ACM 978-1-4503-0942-4/11/10.

Our goal is to produce a small set of recommendations to fix U's misuse of the protocol of the reusable component. For a given analysis system, we must define the following two functions, A, H:

- A such that $A(U, C)$ is a finite set of close alternatives to U which do not break C. A member of $A(U, C)$ is a program which results from a small modification to U and is possibly within protocol.
- H such that if X is a set of alternatives to U correcting C (e.g. $X = A(U, C)$), then $H(X) : X \Rightarrow \mathbb{N}$ is a heuristic which sorts alternatives according to their likelihood of being correct. The benchmark correct alternative(s) is defined to be what a human expert would suggest to correct the broken spec. C.

The final result to the user is the top k alternatives from $H(A(U, C))$. The capabilities of A and H will depend directly upon the analysis system used.

Generating Fusion Suggestions

Fusion provides an expressive specification language to encode arbitrary software protocols, and we define A and H within the context of this analysis system. A brief overview of the system's analysis approach is provided.

Fusion's specification language is based on a first order logic over *relationships*, which are protocol abstractions that indicate meaningful associations between objects upon specific protocol operations. A protocol specification for Fusion defines pre and post conditions upon specific protocol operations, in the form of predicate logic over relationships. By keeping track of the state of relationships throughout the program, Fusion reports a broken specification, along with the offending operation, whenever the precondition is false for that operation.

Given a reusable component which implements a Fusion specification, and a user program U which uses that component, breaking a specification C, we now define A and H.

- $A(U, C)$.
  Define $C(R : RELATIONSHIP \rightarrow \{T, F\}) = TRUE \iff$ the relationships which are T (present) in R satisfy the precondition for C, and let R be the relationships T at the point where C is broken.
  Let $P = \{R\prime | C(R\prime) = true \wedge \forall R\prime\prime . diff(R, R\prime\prime) \subsetneq diff(R, R\prime) \rightarrow C(R\prime\prime) = false\}$, where $diff(A, B) = \{r | A(r) \neq B(r)\}$.
  Hence, of all classes of maps which satisfy constraint C, only the representative which incurs the fewest number of changes from R is in P.
  Define $f(r)$, where
  $f : (RELATIONSHIP \times TRUE, FALSE) \rightarrow MODIFICATION = \{INSERT \cup CREATE\_BRANCH \cup DELETE \cup MOVE \cup \{NONE\}\}$, to be the set of source code modifications that under any precondition of any collaboration constraint have the effect of negating r (union-ed with a no-action option), and let $F(\{r_1 \ldots r_k\}) = f(r_0) \times \ldots f(r_k)$. Hence, F takes a set of relationships which must be negated, and returns all possible combinations of modifications that might have the effect of negating every relationship in the input.
  Then $A(U, C) = \bigcup_{P_i \in P} F(P_i) \cap$ SATISFY_C.
  Hence, $A(U, C)$ is simply the set of all modifications which do result in constraint C being satisfied.
- $H(X)$ The sorting function assigns a score based mainly on whether the alternative does in fact satisfy the constraint C and whether a new broken constraint is introduced, although the types of source code modifications used are also considered.

3. Evaluation

Our system was tested on 23 self contained excerpts of misuse of the file IO, iterator, and asp.net protocols. The following criteria was used to asses the performance of the system's fixes on each misuse example: "expert" if the fix matches the human expert's; "correct" if the fix is within protocol but does not match the human expert's; "wrong" if the fix is not within protocol. A test is judged with the highest grade which appears at least once within the top 3 suggestions. Out of all 23 examples, 14 were judged "expert", 7 were judged "correct" and 2 were judged "wrong".

4. Future Work

Eventually, H could be improved into a learning algorithm based on feedback of human experts over time, providing an indicator of the level of confidence on the correctness for each recommendation.

5. Related Work

[2] focuses on automatically correcting type errors in a program by analyzing similar programs which type-check. This work focuses on type-checking and not on arbitrary protocol conformance, and their approach uses a less focused search than the one described here in order to find feasible alternatives. ERL [3] describes a way to improve auto-generated error messages from a failing logical predicate by concentrating only on the failing atomic predicates. This work does not provide recommendations for fixing errors.

References

[1] C. Jaspan and J. Aldrich. Checking framework interactions with relationships. In Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP 2009)

[2] B. S. Lerner, M. Flower, D. Grossman, Searching for type-error in Programming language design and implementation, 2007

[3] Ciera Jaspan, Trisha Quan, and Jonathan Aldrich. Error Reporting Logic, in the Proceedings of the Conference on Automated Software Engineering, L'Aquila, Italy, 2008.