Clairvoyant Assertions*

Ole Jørgen Abusdal[1], Volker Stolz[1], Violet Ka I Pun[1], Crystal Chang Din[2], and Rohit Gheyi[3]

[1] Western Norway University of Applied Sciences, Norway
{ojab,vsto,vpu}@hvl.no
[2] University of Bergen, Norway
Crystal.Din@uib.no
[3] Universidade Federal de Campina Grande, Brazil
rohit@dsc.ufcg.edu.br

Abstract

Refactorings can subtly change the behavior of software. Developers rely on unit tests to track unanticipated changes of behavior. For some refactorings, additional assertions can be introduced that increase the confidence in the refactored code. However, as this only uncovers problems after they have been introduced, we propose here *clairvoyant assertions* that predict the correctness of a refactoring, specifically, with respect to deadlocks. It is our long-term goal to develop synergies between theorem-proving and runtime checking for software development, where e.g. these assertions coincide with the necessary precondition checks for a refactoring, and can partially be (automatically) discharged or carried forward as runtime checks in case this is not possible.

A refactoring is a program transformation that must preserve the externally observable behavior of a program while improving, by some measure, its internal structure. Refactoring is a well established software engineering practise popularized by Fowler [2].

A possible strategy to check that a refactoring preserves behaviour is to introduce corresponding assertions into the refactored code [1]. These assertions may fire e.g. when executing unit tests and will give the developers feedback on the correctness of the refactoring, albeit only after the fact. We improve on this and avoid wasted effort by proposing to introduce *clairvoyant assertions* in the original code that predict the correctness of the refactoring, and could ideally be discharged by a theorem-prover as part of checking the pre-conditions for the refactoring.

We demonstrate our idea in the context of recent work [3] where refactorings that are straightforward transformations in a sequential setting such as in e.g. sequential Java code become perilous in the concurrent actor-language ABS ("Abstract Behavioral Specification language")[1] where some are shown to introduce deadlocks.

Deadlocks in general and in active-object based languages in particular have traditionally been studied from the perspective of a static analysis. We study deadlocks through a rather natural program transformation that introduces dynamic detection. Observing a refactored program that had been transformed to detect deadlocks we carefully extracted key information from the dynamic detection introduced and used it to slightly alter the dynamic detection applied to the original program in order to now also predict whether a particular refactoring, if applied, would introduce a deadlock; assuming we had no deadlocks any deadlock detection now would mean a detection in the refactored

```
d = p.getDept();
m = d.getManager();
m = p.getManager();
```

(a) call site change

```
interface PersonI {
  DeptI getDept();
  PersonI getManager();
  ...
}
```

(b) interface change

```
class Person(DeptI d)
implements PersonI {
  ...
  PersonI getManager() {
    return d.getManager();
  }
}
```

(c) class change

Figure 1: Hide Delegate refactoring in ABS

*This work is supported by DIKU through the "Modern Refactoring" project.

[1] https://abs-models.org/

program. Let us briefly expland on what the deadlocks are with reference to a particular refactoring applied in ABS.

The refactoring in question, Hide Delegate, can be seen in Fig. 1 where we show the before and after code at the same time by showing what could be the output of a diff; the green parts are added by the refactoring, the red parts are removed. Here a client object c wishes to get a PersonI object through an intermediary DeptI object d that p has access to. The refactoring moves this operation from the call site in c (see fig. 1a) to the object p (see Fig. 1c). For this transformation to be a refactoring it should in a sense remain the same program as far as externally observable changes. The refactoring accomplishes a decoupling; the Client no longer needs to know of the Department. The change caused by the refactoring, which is not seen in and of itself as externally observable, is the different call sequence seen in the sequence diagram shown in Fig. 2. However we do not have enough information from such a sequence diagram whether there will be a problem in ABS.

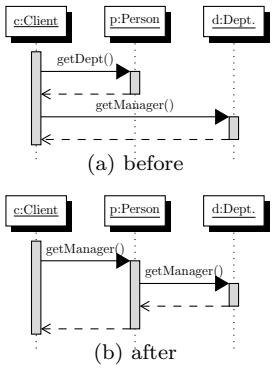

Figure 2: Effect of HideDelegate refactoring

When we also consider the diagram show in Fig. 3 showing additional runtime artifacts of ABS; how objects are partitioned into concurrently running groups in which one object at a time can be "active" while others are blocked. It becomes clear that an execution would result in an attempted call chain as shown in the diagram will produce a deadlock. Here we are showing the objects as circles that are partitioned into squares; the squares represent the concurrent partitioning. A call from c to p will result in c blocking its concurrent group and as such the code in p that wants to call d which resides in the same group as c will remain stuck as d will not become "active" until c releases its hold on the group. We propose a simple transformation that takes a program P and produces a program $T_a(P)$ that is equivalent to P except that where P would deadlock $T_a(P)$ terminates with an assertion. Thus if we have a transformation T_r that should be a refactoring and we know that P does not deadlock, we can get a counterexample of whether $T_r(P) \sim P$ through any observed deadlock termination in $T_a(T_r(P))$. In that case T_r is not a refactoring as the transformation is non-preserving wrt. observable behaviors. The technique of runtime checks is no new result and is seen in other works to ensure safer refactorings [1].

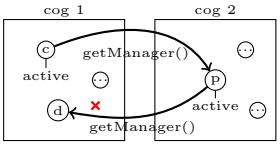

Figure 3: Objects "running" concurrently

A new development is to carefully analyse the assertions in $T_a(T_r(P))$ and when possible reconstruct them in the form of a special transformation T_{ca} such that if we write for a program that executes the steps l and terminates with an assertions $P' \overset{l}{\rightsquigarrow} \bot$ then our special transformation is such that $T_{ca}(P) \overset{l}{\rightsquigarrow}$ iff. $T_a(T_r(P)) \overset{f(l)}{\rightsquigarrow} \bot$ where f is a function mapping steps in $T_{ca}(P)$ to steps in $T_a(T_r(P))$. The assertions in T_{ca} are what we dub *clairvoyant assertions*; predict whether T_r can be safely applied to P. This is not something that we have shown possible in general, but at least for the case of Hide Delegate we can produce such a transformation. It remains to be seen whether other refactorings are admissible for a similar construction of predictive assertions that will fire if the refactoring were to be applied.

References

[1] Anna Maria Eilertsen et al. Safer refactorings. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: (ISoLA'16)*, volume 9952

of *Lecture Notes in Computer Science*, pages 517–531, 2016.

[2] Martin Fowler. *Refactoring - Improving the Design of Existing Code.* Addison Wesley object technology series. Addison-Wesley, 1999.

[3] Volker Stolz et al. Refactoring and active object languages. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'20)*, volume 12477 of *Lecture Notes in Computer Science*, pages 138–158. Springer, 2020.