Regression Verification
Project Application

Dennis Felsing
dennis.felsing@student.kit.edu

September 21, 2013

Abstract

Regression Verification aims at proving the equivalence of two closely related versions of a program, as they often exist in the life cycle of programs. In this project a tool for Regression Verification of programs in a simple while language shall be developed.

1 Problem and Motivation

In *Formal Verification* the correctness of some software is formally proven. For this a formal specification of the correct behaviour of the program has to be given, which poses a difficulty in practice.

Regression Testing on the other hand is the act of testing new versions of some software to discover new bugs. This requires writing extensive test cases, which is laborious and still does not guarantee that all bugs will be found.

The approach of *Regression Verification* is to formally prove that no bugs have been introduced into a new version of some software. This means it is situated between Formal Verification and Regression Testing. Based on two closely related programs, usually an older and a newer version of a program in development, their equivalence shall be proven. Instead of comparing the two programs to a common formal specification Regression Verification tries to make use of their similarity. Therefore neither a formal specification nor test cases are required. Basically the old program version specifies the correct behaviour. This means that the "correctness" that Regression Verification can prove is of a weaker form, as it can only show that a program is as correct as an older version of it.

The purpose of this project is to evaluate new approaches to Regression Verification. This will make Regression Verification applicable to more cases and allow more realistic uses. The approaches will be implemented in a tool for a simple while language.

Kammerjäger is a verification tool for such a simple while language, that has been developed in a Software Engineering Practice project at KIT by Andreas

Eberle, Nicolas Loza, Olga Plisovskaya, Andreas Waidler and Michael Zangl.[1] The parser and interpreter of Kammerjäger will be used to build a new tool for Regression Verification. For this recursions are to be supported in regression proofs. Proofs will be provided by translating the programs to SMT formulas with accompanying Regression Verification rules. The SMT solver Z3 will be used to solve these.

The abstract syntax tree of Kammerjäger will be used as the basis for generating Z3 SMT expressions.

2 Current State of the Art

Regression Verification and variations of it have been subject to many studies.

Godlin and Strichman introduced the term Regression Verification to describe the problem of proving the equivalence of two closely related, successive versions of a program.[2][3] Their approach is simple, using only a single inference rule to prove the equivalence of two related functions. The equivalence of whole programs is then shown by applying this inference rule bottom-up to the call graphs of both programs. A Regression Verification Tool (RVT) for the C language is provided. All loops are transformed to recursions before the proof starts. The use of a bounded model checker as the base of RVT leads to some limitations compared to an SMT solver, for example no further information about recursive functions can be encoded.

Another issue is that for two non-equivalent programs the RVT will not terminate but instead keep trying to prove their equivalance. This approach has recently been expanded to multi-threaded programs.[4]

Hawblitzel et al.[5] provide a framework to encode Regression Verification conditions for a simple While language in Boogie, an intermediate verification language developed at Microsoft Research.

Another approach is taken by Barthe et al.[6] Instead of regressions their focus are optimisations. They generalise the self composition commonly used in non-interference checking to two programs. Unlike the other approaches they explicitly use loops instead of recursions.

Other works deal with proving the equivalence of programs after applying compiler optimisations to them,[7][8] which is different from the user based changes Regression Verification works with.

3 Preliminary Work

An example catalog of 51 pairs of equivalent programs has been compiled from various sources: Related papers, refactoring rules and compiler optimisations. All programs have been translated to the simple while language we use and have been checked for syntactical errors.

In order to assess new approaches some of these programs have been transformed to SMT proof formulas which were then used to verify their equivalence

using Z3. In this multiple approaches were discovered that enable Regression Verification for programs for which Godlin and Strichman's approach failed.

Work on the Regression Verification tool has already been started to facilitate the transformation of programs to SMT formulas for Regression Verification proofs. This will be the basis of the program to be developed in this project.

4 Project Objectives

Our main goals are the following:

1. Develop a tool for Regression Verification for recursive programs in a simple imperative programming language that can be used as the framework for further studies and extensions.

2. Create a case study to evaluate how well our approach for Regression Verification works for different examples in comparison to other approaches.

3. Extend the Regression Verification tool to work with more programs and to be more general.

4.1 Convert loops into recursions

Instead of writing everything using recursive calls loops will also be supported. They will be automatically converted to recursions before running Regression Verification on the programs. This can be done without introducing complexity by extracting the loop bodies to new methods.

The following example programs illustrate the conversion of a loop (left program) to a recursion (right program):

```
1  int min(int[] x, int size) {
2    int i = 1;
3    int m = x[0];
4    while (i < size) {
5      if (x[i] < m) {
6        m = x[i];
7      }
8      i = i + 1;
9    }
10   return m;
11 }
```

```
1  int min(int[] x, int size,
2      int pos) {
3    int m = x[pos];
4    if (pos + 1 < size) {
5      int res = min(x, size,
6                    pos + 1);
7      if (res < m)
8        m = res;
9    }
10   return m;
11 }
```

4.2 Function inlining

Not only calls to other functions, but also recursive calls shall be inlined a limited number of times, meaning the function call is replaced by the function body. This allows proving equivalence in the following examples which was not possible using Strichman's RVT because the abstraction of replacing function

calls with uninterpreted function symbols causes a loss of information which can be avoided by inlining the function instead.

The following programs are equivalent for $\mathbb{N}$. The program on the right side can be obtained by inlining the recursive call to f in line 7 once:

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 1) {
5      r = n;
6    } else {
7      r = f(n - 1);
8      r = n + r;
9    }
10
11   return r;
12 }
```

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 1) {
5      r = n;
6    } else {
7      r = f(n - 2);
8      r = n + (n - 1) + r;
9    }
10
11   return r;
12 }
```

4.3 Finding Counter Examples

Counter examples should be printed for failed proofs if they exist and can be found. For this the inputs to Z3 have to be constructed in a way that allows reconstructing the actual values at the end of a failed proof. These values will then be used to run both procedures to check whether they actually differ or abstraction caused the proof to fail.

The user can fix the bug after seeing the counter example, or, if the difference in behaviour is wanted, use relational equivalence annotations to specify that. Afterwards another regression proof can be started.

4.4 Use solver feedback to determine corner cases

When Z3 tries to prove the following example it will fail by giving its assumed definition of $f(x)$:

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 0) {
5      r = n;
6    } else {
7      r = f(n - 1);
8      r = n + r;
9    }
10
11   return r;
12 }
```

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 1) {
5      r = n;
6    } else {
7      r = f(n - 1);
8      r = n + r;
9    }
10
11   return r;
12 }
```

Note that in line 4 the function bodies differ. This causes the following definition of $f(x)$ to be assumed by Z3:

$$f(x) = \begin{cases} -1 & \text{if } x = 0 \\ g(x) & \text{otherwise} \end{cases}$$

Once we know that the assumption of $f(0) = -1$ is the reason for the failed proof we can execute $f(0)$ as a program to see that the actual return value is 0. This information can then be added to the proof and it will succeed in its next run:

$$f(0) = 0$$

4.5 Functional Condition Extraction

The following is the third and last example Strichman gives for which his method of Regression Verification does not work:

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 1) {
5      r = n;
6    } else {
7      r = f(n - 1);
8
9      r = n + r;
10
11   }
12
13   return r;
14 }
```

```
1  int f(int n) {
2    int r = 0;
3
4    if (n <= 0) {
5      r = n;
6    } else {
7      r = f(n - 1);
8      if (r >= 0) {
9        r = n + r;
10     }
11   }
12
13   return r;
14 }
```

The problem is that in the second function *if (r >= 0)* in line 8 will always be true. But to determine this it has to be known that *f(n-1)* in line 7 will always return a non-negative value. By this information it can be assumed that $f(n) >= 0$. This can be inductively used to prove that $f(n) >= 0$ is true.

In order to extract the condition for *f*, all paths will be traversed and their corresponding path conditions be determined. Out of these the necessary condition for the uninterpreted function of f will be extracted. Now this condition can be checked to be true.

4.6 Relational Equivalence

The goal is not always to prove that two functions return the same values for the same inputs. For this reason the tool should allow specifying a relation in PL1. The default relation will simply be equivalence.

The main use case for this are bug fixes. When a bug was fixed in a function for a particular input the program semantics have been changed deliberately, which can be expressed as a relation.

The following functions calculate the factorial $x!$. The function *fac1* wrongly returns 0 for 0!, whereas *fac2* returns the correct result of 1:

```
1  int fac1(int x) {
2    int r = 0;
3
4
5
6    if (x > 0) {
7      r = x * fac1(x - 1);
8    }
9
10   return r;
11 }
```

```
1  int fac2(int x) {
2    int r = 0;
3
4    if (x == 0) {
5      r = 1;
6    } else if (x > 0) {
7      r = x * fac2(x - 1);
8    }
9
10   return r;
11 }
```

This bug fix can be expressed using the following relation:

$$\forall x : x = 1 \vee fac1(x) = fac2(x)$$

Another use case for relational equivalence are the following two functions to calculate the minumum and maximum of an array of integers respectively:

```
1  int min(int[] x, int s, int p)
       {
2    int m = x[s-1];
3    if (p < s) {
4      m = min(x, s, p + 1);
5      if (x[p] < m) {
6        m = x[p];
7      }
8    }
9    return m;
10 }
```

```
1  int max(int[] x, int s, int p)
       {
2    int m = x[s-1];
3    if (p < s) {
4      m = max(x, s, p + 1);
5      if (x[p] > m) {
6        m = x[p];
7      }
8    }
9    return m;
10 }
```

These program functions satisfy this relation:

$$\forall (int[]x) : max(x) = min(map(\times(-1), x))$$

The user should be able to annotate that the function *max* calculates the maximum while *min* the minimum. That *max* is correct should then be proven based on *min*.

4.7 Creating an example catalog and using it for evaluation

Examples for program transformations have already been collected as a preliminary work. This catalog will be extended with more examples. Possible sources are:

- Other publications (Strichman, Hawblitzel, Kawaguchi, Lopes, Verdoolaege, ...)
- Compiler optimizations (LLVM, GCC, http://www.compileroptimizations.com)

- Refactoring rules (http://www.refactoring.com/catalog/index.html)
- Software-artifact Infrastructure Repository (http://sir.unl.edu/portal/index.php)
- Pex4Fun
- Constructed examples which arise during this project to cover the newly introduced ideas

It shall be evaluated how well our approach works using these examples in comparison to other work

5 Work Plan

5.1 Implement Regression Verification tool

The tool, which will be the basis for further research, has to be developed first.

1. Initially the Abstract Syntax Trees of two programs have to be converted to a corresponding Z3 formula that can be used for Regression Verification. The formula will be true if and only if the two programs can be proven not to be equivalent. Function calls, including recursive ones, will be replaced by uninterpreted function symbols.
2. Another step is the conversion of loops to recursions to allow writing programs with loops. For this loop bodies will have to be extracted to new functions. Special attention to the handling of local variables will be required.
3. The last part of this work package is Function Inlining: Instead of replacing function calls with uninterpreted functions the function body will be inserted to simulate calling the function once. At first the user has to specify how often to inline each function call by annotating the program. Once this is working, automatic exploration of the space of possible unrollings can be implemented.

5.2 Finding Counter Examples

1. In order to find counter examples the SMT input to Z3 has to be constructed in a way that allows reconstructing the values of variables during execution. Specifically this means that an identifier has to be chosen for each variable assignment and that its name must be saved. Alternatively, the identifiers can be chosen in a way that allows the reconstruction of the variable unambiguously from the name.
2. This method shall be used to reconstruct the variable values and use them to symbolically execute the relevant part of the program to determine

whether the two programs actually differ. in which case the user is informed of the input values for which the programs are not equivalent, or that the proof is just incomplete.

5.3 Use solver feedback to determine corner cases

1. When a proof failed the output of Z3 shall be interpreted to determine how functions have been evaluated.

2. These assumptions Z3 made about functions can now be verified to be correct by symbolically executing them using Kammerjäger's interpreter.

3. If an assumption is found to be wrong fixed assumptions about functions for found corner cases shall be added to the corresponding Z3 formula and the proof be repeated.

5.4 Functional Condition Extraction

1. At first it will be necessary to become acquainted with Counterexample-Guided Abstraction Refinement (CEGAR)[9] and determine whether this approach is viable for reverse condition determination as we need it in the case of Regression Verification.

2. Once this is done functional condition extraction can be implemented using CEGAR or by traversing all paths after the function call and determining their path conditions in the functions to be proven equivalent. Determine the necessary condition for the uninterpreted function out of them and encode it in the formula for Z3.

5.5 Relational Equivalence

1. The Regression Verification tool shall be extended to support the specification of a relation in PL1 that specifies how the programs are related to each other instead of just using equality and encode this in the Z3 formula.

2. A concrete syntax shall be defined, so the user can specify these annotations to the program.

5.6 Creating an example catalog and using it for evaluation

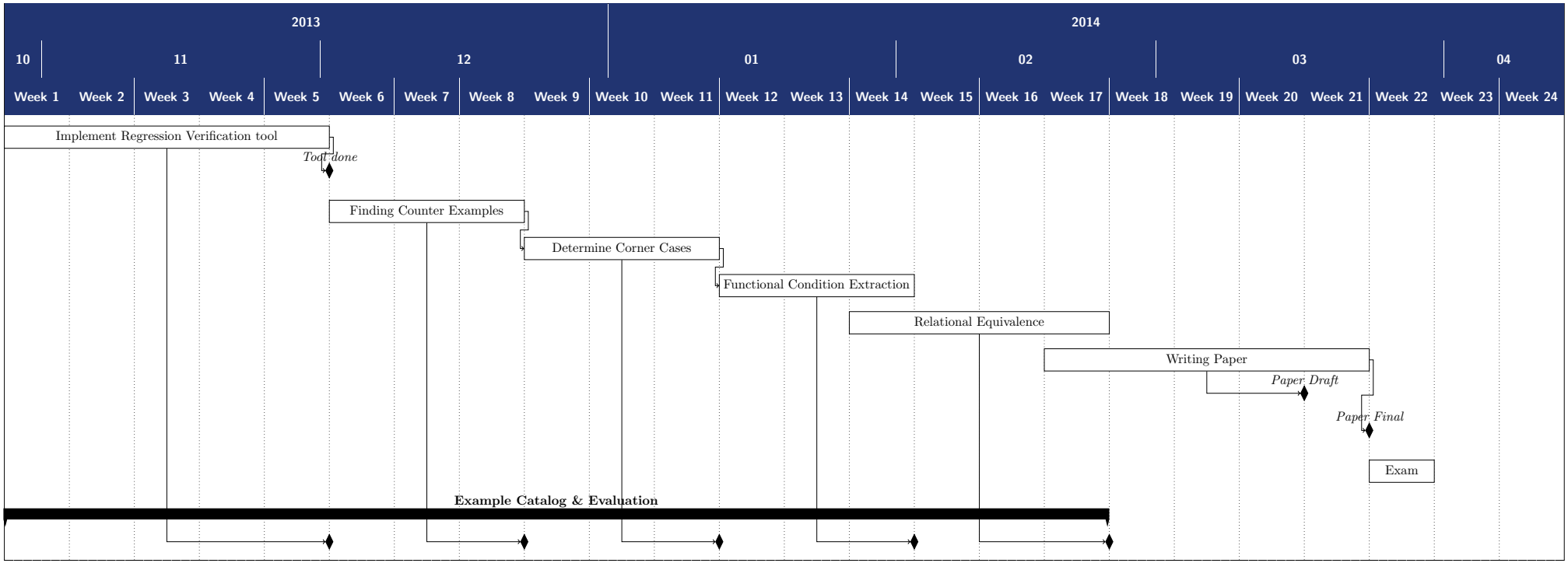
1. More examples shall be added to the existing example catalog based on the sources mentioned in Section 4.5.

2. It is to evaluated which examples can be proven using our approach yet and which are still failing. This information can then be used to determine the reasons they are not working to find out how to evolve our methods.

3. Our results can be compared with other tools for Regression Verification to find out in which cases our approach is more or less successful than the state of the art.

4. Each of the prior work packages shall be evaluated with respect to the database of examples.

6 Time Schedule



Literature

References

[1] "Kammerjäger," http://sourceforge.net/projects/kammerjaeger/, accessed: 2013-09-05.

[2] B. Godlin and O. Strichman, "Regression verification," in *Proceedings of the 46th Annual Design Automation Conference*, ser. DAC '09, 2009, pp. 466–471.

[3] ——, "Inference rules for proving the equivalence of recursive procedures," *Acta Inf.*, vol. 45, no. 6, pp. 403–439, 2008.

[4] S. Chaki, A. Gurfinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *Proceedings of the 13th international conference on Verification, Model Checking, and Abstract Interpretation*, ser. VMCAI'12, 2012, pp. 119–135.

[5] C. Hawblitzel, M. Kawaguchi, S. K. Lahiri, and H. Rebêlo, "Towards modularly comparing programs using automated theorem provers," in *CADE*, 2013, pp. 282–299.

[6] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs," in *Proceedings of the 17th international conference on Formal methods*, ser. FM'11, 2011, pp. 200–214.

[7] L. Zuck, A. Pnueli, B. Goldberg, C. Barrett, Y. Fang, and Y. Hu, "Translation and run-time validation of loop transformations," *Form. Methods Syst. Des.*, vol. 27, no. 3, pp. 335–360, 2005.

[8] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal, "Equivalence checking of array-intensive programs," in *Proceedings of the 2011 IEEE Computer Society Annual Symposium on VLSI*, ser. ISVLSI '11, 2011, pp. 156–161.

[9] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, 2000, pp. 154–169.