Understanding Counterexamples for Relational Properties with *DIbugger*

Mihai Herda[*] Michael Kirsten[†] Etienne Brunner
Joana Plewnia Ulla Scheler Chiara Staudenmaier
Benedikt Wagner Pascal Zwick Bernhard Beckert[‡]
Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
[*]`herda@kit.edu` [†]`kirsten@kit.edu` [‡]`beckert@kit.edu`

Software verification is a tedious process that involves the analysis of multiple failed verification attempts, and adjustments of the program or specification. This is especially the case for complex requirements, e.g., regarding security or fairness, when one needs to compare multiple related runs of the same software. Verification tools often provide counterexamples consisting of program inputs when a proof attempt fails, however it is often not clear why the reported counterexample leads to a violation of the checked property. In this paper, we enhance this aspect of the software verification process by providing *DIbugger*, a tool for analyzing counterexamples of relational properties, allowing the user to debug multiple related programs simultaneously.

1 Introduction

Software verification is a tedious process that involves the analysis of multiple failed verification attempts, and adjustments of the program or specification. Oftentimes, this is an incremental process, where at first neither the formal specification captures the informally-given requirements, nor the program adheres to the specification. The task becomes even trickier when the requirements are complex, as is often the case for security (e.g., noninterference for information flow [13]) or fairness (e.g., for resource allocation [12] or voting [2]) requirements, which can only be captured using *relational properties*.

Relational properties refer to at least two program runs. A classical example of such a property is *program equivalence*—the property that two programs provided with identical inputs generate identical outputs. Relational properties are highly relevant in the field of evolving safety-critical systems, e.g., when modifying the software, or when one software component is replaced with another one. When applying such a change to the software, we wish to make sure that this change does not introduce new bugs. Relational verification tools such as LLRêve [10] can prove that a new—but similar—software program is equivalent (modulo some allowed changes) to the preceding existing software.

In case the verification of these properties fails, existing verification tools can provide counterexamples. Such counterexamples contain concrete inputs which are identical between the two programs, but for which the execution of the two programs leads to two different outputs. Understanding *why* the provided inputs are a counterexample is—however—usually not a trivial task. Whereas this task is already difficult for functional properties, it becomes even more challenging for relational properties, as the user needs to concomitantly check the values of program variables across multiple (i.e., more than one) program runs. Nonetheless, this is a very important step which the user needs to perform in order to improve the analyzed specification and/or code. The process of verifying software is an iterative one, as described in [3] as follows: “Until the verification succeeds, (a) failed attempts have to be inspected in order to understand the cause of failure and (b) the next step in the proof process has to be chosen.”

Contribution. The contribution of this paper is *DIbugger*, a novel tool that supports the user in understanding the reason why some input leads to a violation of a k-relational property. This input may be provided by a verification tool as counterexample for such a property. Thereby, we enhance step (a) in the iterative software verification process mentioned above. DIbugger extends familiar concepts from software debugging in order to support the user in finding the points of execution of the analyzed programs which introduce a violation of the relational property. DIbugger allows for conditional expressions and watch expressions which use conditions and expressions which refer to any or all of the analyzed programs. Moreover, additional user assistance is provided by backwards debugging and adaptable step sizes for each analyzed program. To the best of our knowledge, DIbugger is the first tool that addresses the problem of debugging relational properties.

Structure of the paper. In Section 2, we present DIbugger and its main functionalities. Furthermore, Section 3 shows how DIbugger can be used which is illustrated by applying it on an example. We explain the range of supported properties and present related work in Section 4 and finally conclude in Section 5.

2 DIbugger

DIbugger[1] is a relational debugger for the *WLANG* programming language. WLANG is a subset of the C programming language and supports sequential, interprocedural programs. Dynamic memory allocation and object-oriented programming features are not yet supported.

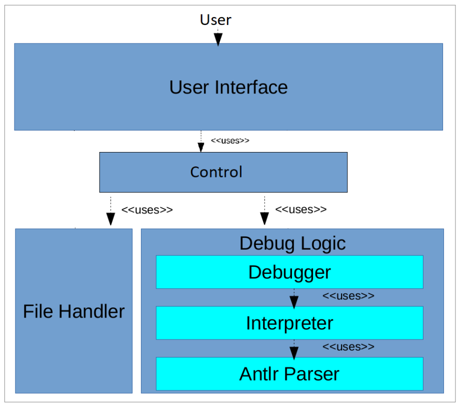
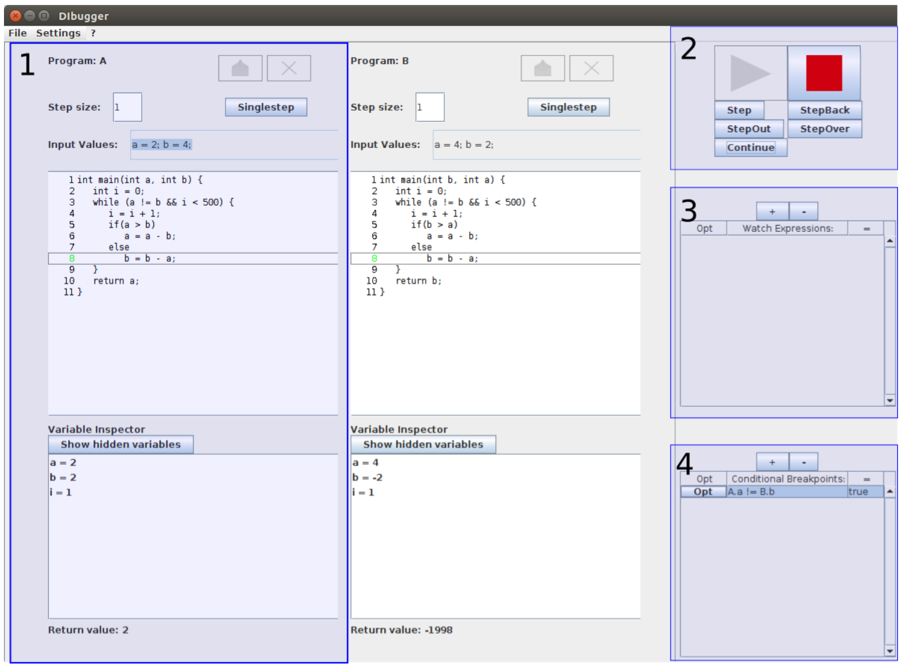


Figure 1: The architecture of DIbugger

As shown in Figure 1, DIbugger consists of four components which are responsible for the user interface, control, file handling and debugging respectively. The debugging functionality is built on top of an interpreter for WLANG. The interpreter generates the trace (i.e., the sequence of values of a program's variables at each point of its execution) of each analyzed program from the given inputs. The debugger works on those traces and executes the debugging operations as selected by the user. The graphical user interface (GUI) of DIbugger is shown in Figure 2, and the available features are explained in the following.

[1]DIbugger is available at `https://git.scc.kit.edu/py8074/dibugger`



Program: A

Step size: 1

Input Values: a = 2; b = 4;

```
int main(int a, int b) {
   int i = 0;
   while (a != b && i < 500) {
      i = i + 1;
      if(a > b)
         a = a - b;
      else
         b = b - a;
   }
   return a;
}
```

a = 2
b = 2
i = 1

Return value: 2

Program: B

Step size: 1

Input Values: a = 4; b = 2;

```
int main(int b, int a) {
   int i = 0;
   while (a != b && i < 500) {
      i = i + 1;
      if(b > a)
         a = a - b;
      else
         b = b - a;
   }
   return b;
}
```

a = 4
b = -2
i = 1

Return value: -1998

Opt	Conditional Breakpoints:	=
Opt	A.a != B.b	true

Figure 2: The user interface of DIbugger

2.1 Debugging Operations

The buttons for the debugging operations are situated in the top right part of the GUI (see Figure 2, in the highlighted area **2**). The *Play* and *Stop* buttons allow for switching between the debug and the edit mode, respectively. In the edit mode, the analyzed programs can be modified, and once the user switches into debug mode, the traces are generated and the programs can be debugged. Moreover, as seen below the *Play* and the *Stop* button, the following buttons provide the specific debugging functionalities:

- *Step*: The execution of each analyzed program advances by the *step size*, i.e., the amount of execution points, specified by the user in the program panel (highlighted area **1** described in Section 2.2). Thereupon, the variable values computed by the instructions in these traversed execution points can be inspected for each program in the *variable inspector* in the program panel. The user can use different step sizes for the analyzed programs in order to both keep the program executions synchronized and examine loops where the numbers of instructions vary between the programs. Depending on the analyzed property or programs, mutable step sizes allow the user to keep the programs in lockstep even when some programs progress faster than others.
- *StepBack*: The execution of each analyzed program moves one step back.
- *StepOut*: The execution of each analyzed program either jumps out of the respective current method or—if already in the outermost method—moves to the end of the main method.
- *StepOver*: The execution of each analyzed program performs a normal step, but does not step into any traversed method call.
- *Continue*: The execution of each analyzed program advances to the next—whichever comes first—breakpoint, conditional breakpoint evaluating to true (see Section 2.3), or end of the main method.

2.2 Program Panels

The central part of the GUI contains one *program panel* for each analyzed program, as illustrated in the highlighted area **1** of Figure 2. The two buttons at the top of the program panel allow the user to add a new program panel or to remove the existing panel. Below these buttons, the user can modify the step size for debugging, or, by using the *singlestep* button, perform a single debugging step only in the selected program. Further down, the user must provide the debugging inputs for the respective program. In the center of the program panel, the user can inspect the analyzed program and set breakpoints anywhere within the code. These breakpoints correspond to synchronization points. When pressing the *Continue* button, all analyzed programs advance to their next breakpoint or—when activated before—conditional breakpoint. Below the analyzed program, the variable inspector shows the current values for all program variables in the scope of the current execution point, as well as the return value of the main method. Note that each program panel has a unique identifier (e.g., the highlighted program panel in Figure 2 has identifier *A*) which the user must use when writing watch expressions and conditional breakpoints (see Section 2.3) referring to the program's variables.

2.3 Watch Expressions and Conditional Breakpoints

When debugging relational properties, the user needs to constantly compare the values of the variables in all analyzed programs. In the context of relational verification, she would need to check whether certain *relational invariants* hold at some points of interest. However, repeating this in every step of the debugging process would be a tiresome task. In order to reduce her effort, she can insert *watch expressions* and *conditional breakpoints* within the highlighted areas **3** and **4** in Figure 2, respectively.

Watch expressions are WLANG expressions which may contain variable identifiers from any of the analyzed programs. They help the user to compare values between the execution points of the analyzed programs. At each point of the debugging process, the value of the expression is computed and the result is displayed. This feature allows the user to check at any time whether certain relational invariants hold.

Conditional breakpoints are boolean expressions which are evaluated at every execution point reached with the step sizes specified by the user. They help the user to find the execution points of her interest. If the expression evaluates to true, then the execution of the analyzed programs halts at that execution point. Conditional breakpoints allow the user to search for execution points in which relational invariants are violated. Using the *Opt* button for both watch expressions and conditional breakpoints allows for setting a program scope. A program scope consists of two line numbers for specifying start and end of the program segment in which the variables in the WLANG expressions are to be evaluated. Thus, if the program execution is outside of the specified scope, the value of the watch expression is unknown and, for conditional breakpoints, the execution does not halt outside of the scope.

3 Using DIBugger on an Example

In the following, we illustrate how DIbugger can be used by applying it on an example for debugging program equivalence in the scope of software evolution. We use two programs, an implementation of Euclid's algorithm for computing the greatest common divisor as shown in Listing 1, and a modified implementation of Euclid's algorithm as shown in Listing 2. The user modifies the original implementation based on the assumption that $gcd(a,b) = gcd(b,a)$ holds. She reversed the condition in the `if`-statement in line 5 such that in line 10 the method returns the value of the variable `b` instead of the value of the variable `a`.

```
 1 int main(int a, int b) {
 2    int i = 0;
 3    while (a != b && i < 500) {
 4       i = i + 1;
 5       if (a > b)
 6          a = a - b;
 7       else
 8          b = b - a;
 9    }
10    return a;
11 }
```

Listing 1: Correct Euclid's Algorithm

```
 1 int main(int b, int a) {
 2    int i = 0;
 3    while (a != b && i < 500) {
 4       i = i + 1;
 5       if (b > a)
 6          a = a - b;
 7       else
 8          b = b - a;
 9    }
10    return b;
11 }
```

Listing 2: Incorrect Euclid's Algorithm

In order to check whether the implementation behaves identically to the original one, a relational verification tool may be used. In our example, the relational verification tool returns a counterexample that consists of an identical input for both programs, e.g., $(a = 2, b = 4)$. With this input, the user can execute the two programs and observe that the original implementation returns 2 and the modified implementation returns -1998. While this shows that the two implementations behave differently with respect to the result value, it does not help the user in understanding *why* this is the case. Therefore, the user needs to debug the two programs in parallel.

When the user starts the debugging process, the return value of each of the two programs is shown below the respective program. This is possible as the interpreter of DIbugger first executes the analyzed programs and then generates their traces. Note that within our application scenario, these computations may not cause any relevant performance problems. Beyond DIbugger's application for understanding counterexamples, the real bottleneck regarding the programs' sizes and complexities lies within the preceding verification task performed by the verification tool.

Generating the trace for each program allows debugging features such as stepping backwards, which is very useful when debugging multiple programs side by side. In case the user performs too many debugging steps and advances beyond the execution point of her interest, she can simply step back in the programs until she reaches her point of interest. With two conventional debuggers—however—she would be required to restart the debugging process. Furthermore, the precomputed traces enable the support of conditional breakpoints, where the user can find pairs of execution points which violate a relational invariant.

In our example, the user can specify that the relational invariant $A.a == B.b$ needs to hold after every debugging step. With this invariant as a conditional breakpoint, pressing the *Continue* button stops the execution in both programs at line 8 with the values 2 and -2 for $A.a$ and $B.b$ respectively. The user then examines these values and understands that she forgot to switch the *then*- and *else*-case in the *if*-statement. Afterwards, she enters the editing mode, edits the second program by applying the necessary changes, and then enters the debugging mode again. Finally, both programs return the (same) value 2.

We see already in this simple example that the analysis using conventional debuggers, which only allows to inspect a single program execution simultaneously, would be more difficult. As in conventional debuggers the user must guide the debugging process for all programs separately, she cannot use watch expressions and conditional breakpoints for finding tuples of execution points which violate the relational property.

4 Supported Properties and Related Work

In the following, we elaborate on two relevant aspects closely related to DIbugger. First, we illustrate the range of (relational) properties which DIbugger supports additionally to the (functional) properties which are also supported by conventional debugging tools. Second, we cover approaches related to DIbugger in the sense that debugging is performed in the scope of or combined with the task of formal verification.

Supported properties. DIbugger supports the inspection of counterexamples for any *k-safety property*, i.e., properties that can be refuted by at most *k* traces [5]. A prominent target for verification of *k*-safety properties is program equivalence. Verification approaches for program equivalence exist, e.g., for C programs [10] or PLC software [4], and allow both verification and counterexample generation. Another example of relational properties are information flow properties which target the problem whether certain outputs can be influenced by certain inputs of the program. The KeY theorem prover [1] supports such properties [13] and can also generate counterexamples. A great variety of relational properties exists, e.g., when specifying fairness properties in the context of social choice theory. Therein, a prominent example are voting algorithms which take the individual votes and compute the elected candidates, with relational properties such as monotonicity, anonymity, neutrality or reinforcement. Relational properties for voting algorithms can also being verified using formal methods [2] and the generated counterexamples to such properties can greatly enhance the understanding and selection of such algorithms [11].

Related work. Debugging itself is a well-known and established technique from software engineering and implemented in a multitude of software development environments. However, we did not find any work on the process of simultaneously debugging multiple programs in a synchronized or relational fashion. One related idea is the concept of *delta debugging*, which searches for failure causes, i.e., how and when the infection causing the software defect has been propagated [6]. Cleve and Zeller attempt to obtain the smallest possible subset of relevant variables by performing a search over both the chain of applied changes and the original variables which might have caused the infection. Thereby, infectious state differences are automatically narrowed down both in time and in space while requiring logarithmic to quadratic runtime.

Another approach more oriented towards understanding counterexamples is the `explain` tool, which works interactively [7]. Groce et al. perform a causal slicing algorithm based on bounded model checking by first producing a counterexample and then computing a successful execution most similar to the failing run using distance metrics. Guided by this distance metrics, the user searches the cause which seems most convincing to her, and finally uses the bounded model checker to verify that the suspected cause is indeed a valid explanation for the failing run. Comparing multiple programs is also interesting for inspecting concurrent programs. Jalbert and Sen apply a greedy slicing technique to simplify complex buggy traces from concurrent program executions to gain a better understanding for the cause of the failure [9].

Moreover, ideas from debugging have also been applied to deductive program verification in order to inspect proofs for program correctness based on logical calculi. Exploiting the technique of symbolic execution, Hentschel et al. devised a symbolic execution debugger which symbolically analyses all possible program states based on the program's formal precondition [8]. This technique allows the inspection of failed proof attempts which help understanding possibly undesired program behaviour. Finally, the debugging mindset can also directly integrated in full program verification on the basis of a compact proof language. Beckert et al. have instrumented theKeY theorem prover for Java programs in order to perform interactive what-if-analyses in a user-friendly fashion directly on the proof object [3]. The user can directly manipulate on the proof using a kind-of proof meta language also allowing to experiment by coming up with and trying out her own assumptions for gaining a detailed understanding on why the proof did not succeed (yet).

5 Conclusion and Future Work

We presented DIbugger, a tool that helps the user understand why the verification of a relational property failed. The tool can be used as a counterexample analyzer for many verification approaches in various scenarios and use cases ranging from regression verification of safety critical system to the verification of information flow properties or the verification of social choice properties.

Moreover, we plan to extend the supported language features to heap-based data structures, and support the automatic suggestion of useful conditional breakpoints or watch expressions, depending on the analyzed relational property. Further ideas to go from here are to enrich DIbugger by property-specific breakpoints in order to better-support specific use cases, or to extend the current breakpoints and watch expressions to quantitative program comparisons. Such ideas could also be integrated in a larger framework guided by counterexamples for abstracting the program. Finally, we would like to apply DIbugger to larger use cases to gain more experiences on its scalability and usability for specific use cases.

References

[1] Wolfgang Ahrendt et al., editors (2016): *Deductive Software Verification - The KeY Book: From Theory to Practice*. *LNCS* 10001, Springer, doi:10.1007/978-3-319-49812-6.

[2] Bernhard Beckert, Thorsten Bormer, Michael Kirsten, Till Neuber & Mattias Ulbrich (2016): *Automated Verification for Functional and Relational Properties of Voting Rules*. In Umberto Grandi & Jeffrey S. Rosenschein, editors: *Sixth International Workshop on Computational Social Choice (COMSOC 2016)*, doi:10.5445/IR/1000092712.

[3] Bernhard Beckert, Sarah Grebing & Mattias Ulbrich (2017): *An Interaction Concept for Program Verification Systems with Explicit Proof Object*. In Ofer Strichman & Rachel Tzoref-Brill, editors: *13th International Haifa Verification Conference on Hardware and Software (HVC 2017)*, *LNCS* 10629, Springer, doi:10.1007/978-3-319-70389-3_11.

[4] Bernhard Beckert, Mattias Ulbrich, Birgit Vogel-Heuser & Alexander Weigl (2015): *Regression Verification for Programmable Logic Controller Software*. In Michael J. Butler, Sylvain Conchon & Fatiha Zaïdi, editors: *17th International Conference on Formal Engineering Methods (ICFEM 2015)*, *LNCS* 9407, doi:10.1007/978-3-319-25423-4_15.

[5] Michael R. Clarkson & Fred B. Schneider (2010): *Hyperproperties*. *Journal of Computer Security* 18(6), doi:10.3233/JCS-2009-0393.

[6] Holger Cleve & Andreas Zeller (2005): *Locating Causes of Program Failures*. In Gruia-Catalin Roman, William G. Griswold & Bashar Nuseibeh, editors: *27th International Conference on Software Engineering (ICSE 2005)*, ACM, doi:10.1145/1062455.1062522.

[7] Alex Groce, Daniel Kroening & Flavio Lerda (2004): *Understanding Counterexamples with `explain`*. In Rajeev Alur & Doron A. Peled, editors: *16th International Conference on Computer Aided Verification (CAV 2004)*, *LNCS* 3114, Springer, doi:10.1007/978-3-540-27813-9_35.

[8] Martin Hentschel, Reiner Hähnle & Richard Bubel (2016): *The Interactive Verification Debugger: Effective Understanding of Interactive Proof Attempts*. In David Lo, Sven Apel & Sarfraz Khur-

shid, editors: *31st International Conference on Automated Software Engineering (ASE 2016)*, ACM, doi:10.1145/2970276.2970292.

[9] Nicholas Jalbert & Koushik Sen (2010): *A Trace Simplification Technique for Effective Debugging of Concurrent Programs*. In Gruia-Catalin Roman & André van der Hoek, editors: *18th International Symposium on Foundations of Software Engineering (SIGSOFT FSE 2010)*, ACM, doi:10.1145/1882291.1882302.

[10] Moritz Kiefer, Vladimir Klebanov & Mattias Ulbrich (2018): *Relational Program Reasoning Using Compiler IR - Combining Static Verification and Dynamic Analysis*. *Journal of Automated Reasoning* 60(3), doi:10.1007/s10817-017-9433-5.

[11] Michael Kirsten & Olivier Cailloux (2018): *Towards automatic argumentation about voting rules*. In Sandra Bringay & Juliette Mattioli, editors: *4ème conférence sur les Applications Pratiques de l'Intelligence Artificielle (APIA 2018)*, doi:10.5445/IR/1000092711.

[12] Tian Lan, David T. H. Kao, Mung Chiang & Ashutosh Sabharwal (2010): *An Axiomatic Theory of Fairness in Network Resource Allocation*. In: *29th International Conference on Computer Communications (INFOCOM 2010)*, IEEE, doi:10.1109/INFCOM.2010.5461911.

[13] Christoph Scheben & Simon Greiner (2016): *Information Flow Analysis*. In: *Deductive Software Verification - The KeY Book: From Theory to Practice*, chapter 13, *LNCS* 10001, Springer, doi:10.1007/978-3-319-49812-6_13.