Modular and Verified Automatic Program Repair

Francesco Logozzo Thomas Ball
Microsoft Research, Redmond
{ logozzo, tball} @microsoft.com

Abstract

We study the problem of suggesting code repairs at *design time*, based on the warnings issued by modular program verifiers. We introduce the concept of a *verified* repair, a change to a program's source that removes bad execution traces while increasing the number of good traces, where the bad/-good traces form a partition of all the traces of a program. Repairs are property-specific. We demonstrate our framework in the context of warnings produced by the modular `cccheck` (a.k.a. `Clousot`) abstract interpreter, and generate repairs for missing contracts, incorrect locals and objects initialization, wrong conditionals, buffer overruns, arithmetic overflow and incorrect floating point comparisons. We report our experience with automatically generating repairs for the .NET framework libraries, generating verified repairs for over 80% of the warnings generated by `cccheck`.

Categories and Subject Descriptors D. Software [*D.1 Programming Techniques*]: D.1.0 General, D.2.1 Requirements/Specifications, D.2.2 Design Tools and Technique, D.2.4 Software/Program Verification D.2.5 Testing and Debugging D.2.6 Programming Environments

General Terms Design, Documentation, Experimentation, Human Factors, Languages, Reliability, Verification.

Keywords Abstract interpretation, Design by contract, Program repair, Program transformation, Refactoring, Static analysis.

1. Introduction

Programs have bugs. Sound static analyzers, such as automatic program verifiers, can catch bugs but usually leave the problem of repairing the program to the developer. During active development, reports of possible bugs may be of little interest to programmers. On the other hand, if a program verifier can suggest code repairs, it can potentially help the programmer write correct code.

We focus on the problem of suggesting code repairs starting from the warning issued by a modular program verifier. A modular verifier uses contracts (*i.e.*, preconditions, postconditions, and object invariants) to decompose the verification problem from the level of a whole program to the level of individual methods. Developer-supplied contracts are essential not only for scalability but also for documenting intent as well as localizing the cause of failures [25].

The first step in addressing the problem is to focus attention on the bugs that matter to developers. In our case, we consider contract violations and runtime errors. A tool like `cccheck` [12], the CodeContracts static checker, can spot those errors at design time. Ideally, `cccheck` should not only report the warnings, but also provide a (set of) possible fixes, that are then presented to the programmer to choose among or reject.

The second step in addressing the problem is to define what a code repair is. Previous work on automatic program repair by Perkins *et al.* [27] defines a repair to be a code transformation such that the repaired program passes a given set of test cases (including the one exposing the bug). This definition is not well-suited to the context of verification and active program development as it requires running the repaired program, which may not always be possible, and it is only as good as the given regression test suite.

Because of these limitations and our desire to statically reason about program defects and repairs, we propose a different definition for repair. A *verified* repair reduces the number of *bad* executions in the program, while preserving, or increasing, the number of *good* runs. A bad run is one that violates a given specification (assertion, precondition, run-time guard, etc.). A good run is one that meets all specifications of the original program. These two sets of traces form a partition of *all the traces* of a program.

Using abstract interpretation, we design a general framework in which code repairs can be expressed. We define several abstractions of trace semantics in order to permit a wide variety of program repairs. One abstraction restricts what is observable about program state to the program points containing assertions, which is necessary since we expect program repairs to change the control flow of programs. A

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

```
void P(int[] a)
{


  for (var i = 0; i < a.Length; i++)
    a[i - 1] = 110;
}
```

```
void P'(int[] a)
{
  Contract.Requires(a != null);

  for (var i = 1; i < a.Length; i++)
    a[i - 1] = 110;
}
```

Figure 1. `Cccheck` detects the possible null-dereference of `a` and a definite buffer underflow in `P`. It suggests the precondition `a != null` and initializing `i` to 1. The resulting correct program is `P'`.

Boolean abstraction further restricts the observations to the Boolean values of assert expressions, which permits changes to program variables appearing in the program and the assertion expression itself. Based on this semantic foundation, one can design different algorithms for code repairs. In general there is no unique recipe for designing verified code repairs, in the same way as in abstract interpretation there is no unique way of designing abstract domains.

We relate the problem of designing code repairs to program verification/analysis as follows. There are three components in program analysis: the program, the semantic knowledge about program behavior, and the property to be verified. In program analysis, the problem is to refine the semantic knowledge so to derive that the properties hold for the program (verification) or do not hold (bug finding). In code repairs, the problem is to *refine the program* using the semantic knowledge so that it satisfies the properties of interest. In `cccheck`, or similar tools, the semantic knowledge is given by the abstract state, belonging to some abstract domain, which was computed by the analysis. The next natural step is to exploit the abstract state to derive the semantic code fix. Therefore, code repairs are *domain-dependent*. We envision code repairs to be standard part of the design of static analyses, abstract domains and verification systems in general.

Main contributions We define the notions of verified repairs in terms of abstractions of trace semantics. We present *sound* algorithms for suggesting program repairs to: contracts (preconditions, postconditions, assumptions, pure methods), initialization (wrong constants, buffer size, object fields), guards (negation, strengthening, weakening), buffer overflows, floating point comparisons, and overflowing expressions. While we use `cccheck` as our verifier, the proposed repairs can be easily adapted and implemented in most any static analyzer.

It is worth noting that our formalization of verified repairs is with respect to the specifications of the original program. As we will see, some repairs are verified by construction. For others, we need to apply the repair and check the repaired program to ensure it does not violate other assertions, *i.e.*, we want to avoid the situation where repairing an assertion causes the failure of another assertion downstream. The fact that all repairs are local to a method means that verifying a repair only requires local analysis and can be applied to incomplete programs. In general, for a given warning, several distinct repairs are possible. Repairs can be ranked according to some metric of interest, *e.g.*, complexity, size of the change, *etc.* We show that the local analysis is very fast – in the order of $150ms$ per method – and that the repair inference is just a tiny portion of that time. We can infer thousands of repairs in large C# libraries and we can propose a repair for more than 80% of the warnings issued by the analyzer. We are confindent that our code repairs can be used in an interactive IDE.

Plan The remainder of the paper is organized as follows. Section 2 illustrates the kind of repairs we are able to produce automatically. Section 3 presents a trace-based semantics of programs. Section 4 defines abstractions over traces and formalizes two kinds of verified repairs. Section 5 extends those abstractions when a static analyzer is used. Section 6 describes the `cccheck` verifier, the types of errors it can find, and the specific repairs enabled by `cccheck`. Section 7 discusses our experience with automatically repairing very large C# programs, Section 8 reviews related work and Section 9 concludes the paper.

2. Examples of Program Repairs

Repair by Contract Introduction. It is often the case that the code of a method is correct only when executed under certain conditions. In the example of Fig. 1, when the parameter `a` is null, there will definitely be a failure due to a null dereference of the parameter. In this case, the `Cccheck` tool suggests the precondition `a != null` using a `Requires` contract [6].

In the example of Fig. 2, when `length` is negative, the array allocation will always fail. In this case, there are two possible repairs: (i) add an explicit assumption `0 <= length`; or (ii) add the postcondition to `GetALength` stating it will always return a non-negative value. `Cccheck` suggests both to the programmer and lets her choose which one to apply. The first repair is useful when `GetALength` is a third-party or external code, as it makes the programmer assumption explicit and prevents `Cccheck` from generating a warning. The second repair documents the behavior of `GetALength`, clearly stating the contract clients of the method can rely upon.

```
int[] ContractRepairs(int index)
{
  var length = GetALength(); // (1)
  var arr = new int[length];
  arr[index] = 9876;
  return arr;
 }
```

Figure 2. Cccheck spots several possible errors in the code (allocation of an array of negative size, buffer overruns). See the paper text for a description the proposed repairs.

```
string GetString(string key)
{
 var str = GetString(key, null);
 if (str == null)
 {
  var args = new object[1];
  args[1] = key; // (*)
  throw new ApplicationException(args);
 }
 return str;
}
```

Figure 3. A (simplified) code snippet from CustomMarshalers.dll. Cccheck detects the buffer overrun at (*), and suggests the allocation of a larger buffer at the line above or changing the index to 0.

In this example, both a buffer underflow and overflow are possible. In these cases, Cccheck proposes the precondition `0 <= index` and the assumption `Assume(index < length)`, making explicit the relationship between the return value of `GetALength` and the parameter `index`.

Repair of Initialization and Off-By-One Errors. Another class of errors arises from the improper initialization of variables (especially loop induction variables) or use of constants just outside a safe zone. In the for loop of Fig. 1, Cccheck detects a buffer underflow and localizes the cause to be in the initialization of `i`. Cccheck infers that: (i) the constraint $0 < \texttt{i}$ should hold on loop entry; and (ii) any positive initial value for `i` removes the buffer undeflow. The initialization $\texttt{i} = 1$ is the one that enables most good runs, and it is therefore the one suggested to the user. In the example of Fig. 3, Cccheck detects a buffer underflow and suggests two potential repairs: allocate a buffer of length at least 2; use 0 to index the array (to avoid the buffer overflow, without introducing an underflow).

Repairing Guards of Conditional Statements. Let us consider the code snippet in Fig. 4, taken from one of the .NET framework libraries. At program point (*): (i) `c != null` should hold, otherwise the program will crash with a null-pointer exception; (ii) cccheck determines that `c` is `null` for all the executions reaching that point (a definite error).

```
void ValidateOwnerDrawRegions(
  ComboBox c, Rectangle updateRegionBox)
{
  if (c == null)
  {
    var r = new Rectangle(0, 0, c.Width); // (*)
   // use r and c
  }
}
```

Figure 4. A (simplified) snippet from a bug found in System.Windows.Forms.dll. cccheck proposes the repair `c != null` as a precondition or as a replacement for the guard of the if-then statement.

```
IMethodCallMessage ReadArray(
   object[] callA, object handlerObject)
{
  if (callA == null) return null;
  var num = 0;
  if (NonDet())) num++;
  if (callA.Length < num)
    throw new SerializationException();

  // here callA.Length >= num

  this.args = (object[])callA[num++];
   // ...
}
```

Figure 5. A (simplified) snippet from a bug found mscorlib.dll. Cccheck proposes to change the guard to `callA.Length <= num`

Cccheck suggests three repairs: the *necessary* precondition `c != null`; flipping the guard from `c == null` to `c != null`; removing the branch altogether. Note that neither repair removes any *good* trace present in the original program, but does remove bad traces. Also, there is no *best* repair. We do not rank repairs: we simply report them at the program point where they should be applied.

In Fig. 5, cccheck suggests to strengthen the `if`-guard to the condition `callA.Length <= num`, as a buffer overflow may happen otherwise (a `Serialization` exception is not considered an error).

Repairing Erroneous Floating Point Comparisons. Floating point comparisons may produce unexpected results [23]. The .NET semantics enforces the runtime to use: (i) for stack values, the most precise floating point representation available from the hardware; (ii) for heap values, the representation exactly matching the nominal type. In the code of Fig. 6, the parameter `d0` may be a very small, non-zero double represented by 80 bits on `x86`. The test succeeds, but the next assignment causes the truncation of the value of `d0` to a 64-bit quantity, that may be zero, violating the object

```
class FloatingPoint
{
 double d;

 [ContractInvariantMethod]
 void ObjectInvariant()
 {
  Contract.Invariant(this.d != 0.0);
 }

 public void Set(double d0)
 {
  // here d0 may have extended double precision
  if (d0 != 0.0)
    this.d = d0; // d0 can be truncated to 0.0
  }
}
```

Figure 6. A (simplified) snippet from a bug found `mscorlib.dll`. The store field causes the truncation of `d0` which may break the invariant, despite the guard. `Cccheck` proposes repairing the guard by adding the truncation to `d0`.

```
int BinarySearch(int[] array, int value)
{
  Contract.Requires(array != null);
  int inf = 0, sup = array.Length - 1;

  while (inf <= sup)
  {
   var index = (inf + sup) /2 ; // (*)
   var mid = array[index];

   if (value == mid) return index;
   if (mid < value) inf = index + 1;
   else             sup = index - 1;
  }
  return -1;
}
```

Figure 7. `Cccheck` detects and automatically proposes a repair for overflow in the computation of the variable `mid`, using the loop invariants automatically inferred by the abstract interpreter.

invariant. `Cccheck` identifies the problem and suggests repairing the guard to `(double)d0 != 0.0`, *i.e.*, forcing the comparison of the 64-bit truncation of `d0` to zero.

Repairing Overflowing Expressions. Verified repairs are very helpful for dealing with unintended arithmetic overflows. Consider the classical binary search example of Fig. 7: The expression at `(*)` may overflow, setting `index` to a negative value, causing a buffer underflow in the next line. `Cccheck` suggests repairing the expression to `inf + (sup - inf) / 2`, which: (i) allows more good runs (inputs that previously caused the overflow now are ok); (ii) is based on the loop invariant automatically discovered by `cccheck`: $0 \leq \texttt{inf} \leq \texttt{sup} < \texttt{array.Length}$.

```
void ThreadSafeCopy(char* sourcePtr, char[] dest,
                    int destIndex, int count)
{
 if (count > 0)
   if ((destIndex > dest.Length)
    || ((count + destIndex) > dest.Length))
     throw new ArgumentOutOfRangeException();
   { // ... }
}
```

Figure 8. A code snippet from a bug in `mscorlib.dll`. `Cccheck`detects that `count + destIndex` may overflow and suggests repairing the expression to the non-overflowing `count > dest.Length - destIndex`.

In the example of Fig. 8, `count` can be a very large positive value, causing `count + destIndex` to overflow. `Cccheck` suggests repairing the expression to `count > dest.Length - destIndex`.

3. Trace Semantics

We formalize the notion of a verified program repair via a trace semantics. As we are only interested in repairs of violations of safety properties, we only consider finite traces.

Let `P` be a program. `P(pc)` denotes the statement at program point `pc`, and $\texttt{P}[\texttt{pc} \mapsto \texttt{S}]$ denotes the program that is the same as `P` everywhere except `pc`, where it contains the statement `S`. If `S` is a compound statement, a remapping of `S`'s program points may be needed. We keep the remapping implicit to simplify the notation. We let $\mathbb{E}$ denote the set of pure Boolean expressions.

Let Σ be a set of states, and $\tau_{\texttt{P}} \in \wp(\Sigma \times \Sigma)$ be a non-deterministic transition relation. For a state $s \in \Sigma$, $s(\texttt{C})$ denotes the basic command associated with the state, *e.g.*, an assignment, an assumption, or an assertion. The set of blocking states, *i.e.*, states with no successors, is $\mathfrak{B} = \{s \in \Sigma \mid \forall s'.\ \neg\tau_{\texttt{P}}(s, s')\}$. The set of erroneous state, *i.e.*, states violating some assertion `e`, is $\mathfrak{E} = \{s \in \Sigma \mid s(\texttt{C}) = \texttt{assert e} \wedge s \not\models \texttt{e}\} \subseteq \mathfrak{B}$.

Traces are sequences of states. Concatenation is denoted by juxtaposition and extended to sets of traces. $\vec{\Sigma}^n$ is the set of non-empty *finite traces* $\vec{s} = \vec{s}_0 \ldots \vec{s}_{n-1}$ of length $|\vec{s}| = n \geq 0$ including the *empty trace* $\vec{\epsilon}$ of length $|\vec{\epsilon}| \triangleq 0$. $\vec{\Sigma}^+ \triangleq \bigcup_{n \geqslant 1} \vec{\Sigma}^n$ is the set of *non-empty finite traces* and $\vec{\Sigma}^* = \vec{\Sigma}^+ \cup \{\vec{\epsilon}\}$. The set of *finite bad traces*, *i.e.*, traces containing an error is $\vec{\mathfrak{E}} = \{\vec{s} \in \vec{\Sigma}^+ \mid \exists i \in [0, |\vec{s}|).\ s_i \in \mathfrak{E}\}$. The bad (resp. good) traces of $T \subseteq \vec{\Sigma}^*$ are $\mathcal{B}(T) \triangleq T \cap \vec{\mathfrak{E}}$ (resp. $\mathcal{C}(T) \triangleq T \cap (\vec{\Sigma}^* \setminus \vec{\mathfrak{E}})$). The function $\mu \in \wp(\Sigma^*) \to \wp(\Sigma^*)$ filters the maximal traces out of a set of traces. Given a set of traces T, $\mu(T)$ is the largest set satisfying the properties: $\mu(T) \subseteq T$ and $\forall \tau \in \mu(T).\ \nexists \tau' \in T. \exists \tau''. \tau'' \neq \vec{\epsilon} \wedge \tau' = \tau\tau''$.

The *maximal execution traces* or *runs* are prefix traces generated by applying the transition relation from the initial states until a fixpoint is reached (partial execution traces) followed by a projection on the maximal traces:

$$\vec{\tau}^{+}_{\mathtt{P}}(S) = \mu(\mathsf{lfp}\lambda T.\ S \cup \{\sigma_0 \dots \sigma_n\sigma_{n+1} \mid \sigma_0 \dots \sigma_n \in T \wedge \tau(\sigma_n, \sigma_{n+1})\}). \quad (1)$$

The *bad (resp. good) finite complete runs*, or simply *bad (resp. good) runs* of P are $\mathcal{B}_{\mathtt{P}} \triangleq \mathcal{B}(\vec{\tau}^{+}_{\mathtt{P}})$ (resp. $\mathcal{G}_{\mathtt{P}} \triangleq \mathcal{G}(\vec{\tau}^{+}_{\mathtt{P}})$).

The definitions above (and in the following) can be extended as in [3] to take into account infinite runs, but we avoid doing it here to keep the presentation as simple as possible.

4. Verified Repairs

Hereafter, we assume that P is the program containing a bug and P' the repaired version. Intuitively, P' should have more good runs and fewer bad runs than P. Because a repair may change the program's control flow, introducing new states, and possibly new assertions, the concrete traces of P and P' may appear very different. Thus, the simple inclusions $\mathcal{G}_{\mathtt{P}'} \supseteq \mathcal{G}_{\mathtt{P}}$ and $\mathcal{B}_{\mathtt{P}'} \subseteq \mathcal{B}_{\mathtt{P}}$ may be too strict and hold only for trivial repairs.

Instead, we compare the semantics of P and P' at a higher level of abstraction. First, we remove all states but those containing assertion statements (the *assertion trace semantics*). Then, we remove all new assertions introduced in P'. Abstract interpretation [4] provides the right framework to formalize this.

Basic Elements of Abstract Interpretation. We first recall some basic facts and notations about abstract interpretation. A *Galois connection* $\langle L, \leqslant\rangle \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \langle \overline{L}, \sqsubseteq\rangle$ consists of posets $\langle L, \leqslant\rangle$, $\langle \overline{L}, \sqsubseteq\rangle$ and maps $\alpha \in L \to \overline{L}$, $\gamma \in \overline{L} \to L$ such that $\forall x \in L, y \in \overline{L} : \alpha(x) \sqsubseteq y \iff x \leqslant \gamma(y)$. In a Galois connection, the *abstraction* α preserves existing least upper bounds (lubs) hence is monotonically increasing so, by duality, the *concretization* γ preserves existing greatest lower bounds (glbs) and is monotonically increasing. The composition of Galois connections is a Galois connection.

Assertion Trace Semantics. The assertion abstraction $\alpha_{\mathtt{A}}$ removes all states but those referring to assertions. The abstraction $\alpha^1_{\mathtt{A}} \in \vec{\Sigma}^* \to \vec{\Sigma}^*$ on a single trace:

$$\alpha^1_{\mathtt{A}}(\vec{s}) = \begin{cases} \vec{\epsilon} & \vec{s} = \vec{\epsilon} \\ s\alpha^1_{\mathtt{A}}(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) = \texttt{assert e} \\ \alpha^1_{\mathtt{A}}(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) \neq \texttt{assert e} \end{cases}$$

can be lifted to a set of traces $\alpha_{\mathtt{A}} \in \wp(\vec{\Sigma}^*) \to \wp(\vec{\Sigma}^*)$: $\alpha_{\mathtt{A}}(T) = \bigcup_{\vec{s}\in T} \alpha^1_{\mathtt{A}}(\vec{s})$. The function $\alpha_{\mathtt{A}}$, is a complete $\cup$-morphism. Thus, it exists a unique concretization $\gamma_{\mathtt{A}}$ such that $\langle \wp(\vec{\Sigma}^*), \subseteq\rangle \underset{\alpha_{\mathtt{A}}}{\overset{\gamma_{\mathtt{A}}}{\leftrightarrows}} \langle \wp(\vec{\Sigma}^*), \subseteq\rangle$ [4]. The *assertion trace semantics* of P is $\alpha_{\mathtt{A}}(\vec{\tau}^{+}_{\mathtt{P}})$.

In general, a repair may introduce new assertions (that may or may not hold). As the goal of a repair is to address failing assertions of the original program, we remove from the assertion semantics of P' all the *new* assertions and the *new* variables before comparing the behaviors of P and P'.

Let $\delta_{\mathtt{P},\mathtt{P}'}$ denote a *repair* that transforms program P to program P' and let $\mathbb{A}(\delta_{\mathtt{P},\mathtt{P}'})$ be all the *new* assertions introduced by the repair in P'. Let $\pi_{\delta_{\mathtt{P},\mathtt{P}'}} \in \Sigma \to \Sigma$ denote the state projection over all the common variables of P and P'. The function $\alpha^1_{\delta_{\mathtt{P},\mathtt{P}'}} \in \vec{\Sigma}^* \to \vec{\Sigma}^*$ removes all the new assertions and new variables from a trace:

$$\alpha^1_{\delta_{\mathtt{P},\mathtt{P}'}}(\vec{s}) = \begin{cases} \vec{\epsilon} & \vec{s} = \vec{\epsilon} \\ \pi_{\delta_{\mathtt{P},\mathtt{P}'}}(s)\alpha^1_{\delta_{\mathtt{P},\mathtt{P}'}}(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) \notin \mathbb{A}(\delta_{\mathtt{P},\mathtt{P}'}) \\ \alpha^1_{\delta_{\mathtt{P},\mathtt{P}'}}(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) \in \mathbb{A}(\delta_{\mathtt{P},\mathtt{P}'}) \end{cases}$$

and its lifting to sets $\alpha_{\delta_{\mathtt{P},\mathtt{P}'}} \in \wp(\vec{\Sigma}^*) \to \wp(\vec{\Sigma}^*)$, defined as $\alpha_{\delta_{\mathtt{P},\mathtt{P}'}}(T) = \bigcup_{\vec{s}\in T} \alpha^1_{\delta_{\mathtt{P},\mathtt{P}'}}(\vec{s})$, is a complete $\cup$-morphism, so that it exists a concretization function $\gamma_{\delta_{\mathtt{P},\mathtt{P}'}}$ such that $\langle \wp(\vec{\Sigma}^*), \subseteq\rangle \underset{\alpha_{\delta_{\mathtt{P},\mathtt{P}'}}}{\overset{\gamma_{\delta_{\mathtt{P},\mathtt{P}'}}}{\leftrightarrows}} \langle \wp(\vec{\Sigma}^*), \subseteq\rangle$.

Verified Repairs. We are now ready to formally define the concepts of a *verified repair* and a repaired program *improving* another program.

DEFINITION 1 (Verified repair, improvement). *If $\alpha_A(\mathcal{G}_P) \subseteq \alpha_{\delta_{P,P'}} \circ \alpha_A(\mathcal{G}_{P'})$ and $\alpha_A(\mathcal{B}_P) \supseteq \alpha_{\delta_{P,P'}} \circ \alpha_A(\mathcal{B}_{P'})$, then we say that $\delta_{P,P'}$ is a verified repair for P and that P' is an improvement of P.*

The above definition denies the identity (*i.e.*, program P itself) as a trivial improvement, since the number of bad traces should strictly decrease. It allows the removal of an *always* failing assertion as a repair. If an assertion fails in some executions and passes in others, then its removal is disallowed (as the subset inclusion on good runs will fail). For a given program P, there may be several distinct improvements $\mathtt{P'}_1, \mathtt{P'}_2 \dots$ (*e.g.*, Fig. 2 or Fig. 4). One can define an additional scoring function to rank $\mathtt{P'}_1, \mathtt{P'}_2 \dots$. The definition of verified repair naturally induces a partial order on programs (and hence on improvements): a program Q improves R, written $\mathtt{R} \sqsubseteq \mathtt{Q}$ if $\delta_{\mathtt{R},\mathtt{Q}}$ is a verified repair for R. We only compare the "same" assertions over two versions of the program, so P' may introduce new bugs, which requires a new code fix, *etc.* The code fixing process can be iterated to a fixpoint. In general, the least fixpoint may not exist (more hypotheses are needed).

The definition above requires not only all the assertions to be the same, but also that the variables have the same concrete values. We can relax this by introducing a further abstraction $\alpha^1_t \in \vec{\Sigma} \to \wp(\Sigma_a)$, $\Sigma_a \triangleq \epsilon \cup \{\mathtt{true}, \mathtt{false}\} \times \mathbb{E}$, which abstracts from a state everything but the assertion and its truth value. Furthermore α^1_t forgets the execution order –

it only focuses on the truth value of the assertions:

$$\alpha^1_t(\vec{s}) = \begin{cases} \emptyset & \vec{s} = \vec{\epsilon} \\ \{\langle b, \mathtt{e}\rangle\} \cup \alpha^1_t(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) = \texttt{assert e} \\ & \quad \wedge\, b = s \models \mathtt{e} \\ \alpha^1_t(\vec{s'}) & \vec{s} = s\vec{s'} \wedge s(\mathtt{C}) \neq \texttt{assert e} \end{cases}$$

The lifting to sets of traces $\alpha_t \in \wp(\vec{\Sigma}^*) \to \wp(\Sigma_a)$, defined as $\alpha_t(T) = \bigcup_{\vec{s}\in T} \alpha^1_t(\vec{s})$, is a complete $\cup$-morphism, so that it exists a concretization function γ_t such that $\langle \wp(\vec{\Sigma}^*), \subseteq\rangle \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} \langle \wp(\Sigma_a), \subseteq\rangle$.

DEFINITION 2 (Verified assertion repair, assertion improvement). *If* $\alpha_t \circ \alpha_A(\mathcal{G}_P) \subseteq \alpha_t \circ \alpha_{\delta_{P,P'}} \circ \alpha_A(\mathcal{G}_{P'})$ *and* $\alpha_t \circ \alpha_A(\mathcal{B}_P) \supseteq \alpha_t \circ \alpha_{\delta_{P,P'}} \circ \alpha_A(\mathcal{B}_{P'})$*, then we say that* $\delta_{P,P'}$ *is a verified* assertion *repair for* P *and that* P' *is an* assertion *improvement for* P*.*

Thus, an assertion improvement P' focuses on the assertion behavior, guaranteeing that: (i) the repair decreases the number of assertions violated; (ii) no regression is introduced. A verified assertion repair is a weaker concept than verified repair, as it allows the addition of new traces that change the behavior of the program while not breaking the old assertions. We say that a program Q $a-$improves R, written R $\sqsubseteq_a$ Q if $\delta_{\mathtt{R},\mathtt{Q}}$ is verified assertion repair for R.

Here we are interested in repairing failing assertions. However, one can think of modifying the Definition 2 to focus on other behaviors to fix, *e.g.*, memory allocation, resources usage, or performance.

5. Program Repairs from a Static Analyzer

When the state space Σ is finite, the fixpoint equation (1) can be exactly computed and the abstractions and the definitions in Section 4 can be applied as they are. The state space can be made finite either by requiring the programmer to provide loop invariants (as in deductive verification [15]) or by fixing ahead of time a *finite* set of predicates to be used (as in predicate abstraction [2]).

In practice, state space finiteness is too strong a requirement. We want to avoid it, unlike [14, 30]. For instance, under the finiteness hypotheses we cannot *automatically* provide a repair for the overflowing expression in Fig. 7. The repair is based on the knowledge of the loop invariant $0 \leq$ `inf` $\leq$ `sup` $<$ `array.Length` (to prove that `sup` $-$ `inf` does not underflow). In general to infer such an invariant the abstract domain should at least include the abstract domain of intervals [4], which is of infinite width and height.

An abstract interpretation-based static analyzer, like `cccheck`, does not make the finite states hypothesis. It computes an over-approximation of the trace semantics of P [1]:

$$\vec{\tau}^+_{\mathtt{P}} \subseteq \gamma(\mathtt{cccheck(P)}). \tag{2}$$

[1] The concretization function γ maps `cccheck` abstract domains into concrete execution traces. In general there is no best abstraction α.

```
int NotDecidable(int x)
{
  string s = null;
  if(P(x))
    s = "Hello␣world";
  return s.Length;
}
```

Figure 9. An example showing the undecidability of code repairs. The predicate P is `true` for each `x`, but it cannot be decided.

Our goal is to exploit the information gathered by a `cccheck`-like tool to automatically suggest verified repairs for incomplete programs. We let

$$\overline{\mathcal{B}}_{\mathtt{P}} \triangleq \mathcal{B}(\gamma(\mathtt{cccheck(P)}))$$

(resp. $\overline{\mathcal{G}}_{\mathtt{P}} \triangleq \mathcal{G}(\gamma(\mathtt{cccheck(P)}))$) denote the *inferred* bad runs (resp. good runs) of P. Definition 1 and Definition 2 can immediately be extended to use $\overline{\mathcal{B}}_{\mathtt{P}}$ and $\overline{\mathcal{G}}_{\mathtt{P}}$ instead of $\mathcal{B}_{\mathtt{P}}$ and $\mathcal{G}_{\mathtt{P}}$. Because of over-approximation, it may be the case that more bad traces are inferred than the ones in the program's concrete semantics. In practice, this means that sometimes `cccheck` cannot detect that an assertion is always satisfied in the concrete, and it suggests a repair for it, to shut off the warning. Nevertheless, this is not a problem for us, for three main reasons. First, verified code repairs are supposed to be used as design-time suggestions in the IDE: the programmer has the last word on whether or not to apply a code repair. Second, the code repair generated from a false warning helps the programmer understanding the nature of the alarm. Third, by construction, verified repairs improve the program in the sense of Definition 1 or Definition 2. If the repair is such that P $\sqsubseteq$ P$'$, then it introduces in P' only checks that *at worst* are redundant with those that were already present in P. Otherwise, if the repair satisfies P $\sqsubseteq_a$ P$'$, the repair is guaranteed to not violate assertions that were (proven) definitely valid in P.

EXAMPLE 1. Let us consider the example in Fig. 9. The predicate P is such that $\forall x.\mathtt{P}(x) = \mathtt{true}$, but it cannot be decided. Such a predicate exists because of Gödel's incompleteness theorems. Therefore, the `s` dereference cannot be proven, and two code fixes can be suggested, satisfying respectively Definition 1 and Definition 2: (i) the (trivial) addition of the assumption `s != null` just before the return statement, adding a *redundant* check at runtime; (ii) the initialization `s = ""`. Both repairs will stop `cccheck` from reporting the warning.

6. Program Repairs in Practice

Actual verified repairs are property-specific. They exploit the inferred semantic information and the specification (in the form of contracts or runtime errors) to automatically pro-

duce a program repair. We infer program repairs by leveraging: (i) a backwards *must* analysis to propose new contracts, initializations, and guards; (ii) a forward *may* analysis to propose off-by-one, floating point comparisons, and arithmetic overflows code repairs.

6.1 Clousot

We extended `cccheck`, an abstract interpretation-based static analyzer for .NET [12], to generate verified repairs. `Cccheck` has four main phases: (i) assertion gathering; (ii) fact inference; (iii) proving; (iv) report warnings and suggest repairs. In the first phase, `Cccheck` gathers the program assertions, either provided by the programmer, *e.g.*, as contracts, or by language semantics, *e.g.*, division by zero, null pointer, *etc.* Then, it uses abstract interpretation to infer facts about the program.

`Cccheck` includes abstract domains for heap abstraction [10], nullness checking [11], scalable numerical analysis [18, 22], universally and existentially quantified properties [5], and floating point comparisons. [23] `Cccheck` uses the inferred facts to discharge the gathered assertions.

`Cccheck`'s decision procedure has four possible outcomes: (i) *true*, the assertion holds for all executions reaching it, if any; (ii) *false*, every execution reaching the assertion, if any, will cause it to fail (*e.g.*, Fig. 4); (iii) *bottom*, no execution will ever reach the assertion; (iv) *top*, we do not know, as the assertion may be violated sometimes or the analysis was too imprecise. If the outcome is *top* or *false*, `Cccheck` tries to find a verified repair before reporting the warning/error to the user. If a verified repair is found (in general there may be more than one repair for a warning) then: (i) it is reported to the user via a graphical interface; and (ii) it is used by the warning scoring algorithm to produce a ranking of warning (*e.g.*, a warning with a verified repair gets a higher score than a warning without a verified repair).

The above outcomes give an easy algorithm to check whether `P` $\sqsubseteq_a$ `P'`: check (for the matching asserts) if `cccheck` reports fewer *top* and *false* for `P'` than `P` without reducing the number of the *true* results. Correctness follows by the analyzer soundness property (2). In practice re-analysis is not a big problem: on average a method is analyzed in $156ms$.

6.2 Repairs Inferred by Backwards Analysis

The precondition inference of [6] is a goal-directed backward analysis $\mathbb{B}_{\texttt{pc}}(\texttt{e})$, starting from a failing assertion `e`. For each program point `pc`, if $\mathbb{B}_{\texttt{pc}}(\texttt{e})$ does *not* hold at `pc`, then `e` will *necessarily* fail later in the program. The expression $\mathbb{B}_{\texttt{pc}}(\texttt{e})$ is a necessary condition for `e` at `pc`. We omit here the details of $\mathbb{B}$, leaving it as a parameter. Different choices are possible, enabling a fine tuning of the precision/cost ratio. In general, $\mathbb{B}$ is an under-approximation of the semantics, computing fixpoints when loops are encountered [2]. The advantage of using a necessary condition is that we are guaranteed not to remove any good trace. We use the analysis $\mathbb{B}$ to suggest repairs, by matching $\mathbb{B}_{\texttt{pc}}(\texttt{e})$ and the statement $\texttt{P}(\texttt{pc})$ as follows.

Repair by Contract. Contracts (preconditions, postconditions, object invariants, assertions and assumptions) are used for code documentation and by the static checker to perform the assume/guarantee reasoning. The backward analysis can be used to suggest contracts.

Definition 1 generalizes the precondition inference problem of [6]. As a consequence, the inference of *necessary* preconditions is a form of verified repair. A candidate necessary *pre*condition is $\mathbb{B}_{\texttt{entry}}(\texttt{e})$. If $\mathbb{B}_{\texttt{entry}}(\texttt{e})$ meets the visibility and inheritance constraints of the enclosing method [20], then it can be suggested as precondition. Otherwise it is suggested as an assumption or a candidate object invariant. In both cases $\texttt{P} \sqsubseteq \mathbb{B}_{\texttt{entry}}(\texttt{e})$; `P` follows from the fact that $\mathbb{B}$ only produces necessary conditions (hence cutting bad runs).

It may be the case that the backwards analysis stops at $\texttt{pc} \neq \texttt{entry}$, *i.e.*, before reaching the entry point of the method. For instance, this happens when variable in the goal expression is the return value from a method. We can still suggest a repair, either in the form of an `Assume` or as a candidate postcondition for the callee. No good trace is removed: $\texttt{P} \sqsubseteq \texttt{P}[\texttt{pc} \mapsto (\texttt{P}(\texttt{pc}); \texttt{Assume}(\mathbb{B}_{\texttt{pc}}(\texttt{e})))]$.

EXAMPLE 2. For the array store in Fig. 2, `Cccheck` collects the two assertions $0 \leq \texttt{index}$ and $\texttt{index} < \texttt{arr.Length}$. The inferred facts are not sufficient to prove that the assertions will not fail, so `Cccheck` propagates the assertions backwards. For the first assertion, $\mathbb{B}_{\texttt{entry}}(0 \leq \texttt{index}) = 0 \leq \texttt{index}$ is suggested as precondition (`index` is a parameter). The precondition is necessary, because if `index < 0` then a buffer underflow definitely will occur. The precondition is not sufficient, as it does not guarantee that the array indexing is in bounds.

For the second assertion, $\mathbb{B}_{\texttt{entry}}(\texttt{index} < \texttt{arr.Length}) = \texttt{true}$, but $\mathbb{B}_{(1)}(\texttt{index} < \texttt{arr.Length}) = \texttt{index} < \texttt{length}$ must hold between the return value of `GetALength` and the method parameter, otherwise the program will necessarily fail. Therefore, `Cccheck` suggests to make the assumption explicit using the repair `Contract.Assume` $(\texttt{index} < \texttt{length})$. For the array creation, the safety condition $0 \leq \texttt{length}$ cannot be proven either, and the cause is traced back to the value returned by `GetALength`. Two repairs are possible: either an assumption (to document it) or a postcondition for `GetALength`.

Initialization. The necessary condition analysis $\mathbb{B}(\texttt{e})$ can be used to infer repairs for initialization and guards. Let k be a compile-time constant and $\texttt{i}=k$ the statement at the pro-

[2] Please note that $\mathbb{B}$ is *not* Dijkstra's wp predicate transformer. The weakest precondition requires the program to be correct on all the possible paths, no matter which non-deterministic choice is made.

gram point pc. If $\mathbb{B}_{\texttt{pc}}(\texttt{e}) = \texttt{i=}k'$, with $k' \neq k$, then we have detected an erroneous initialization. We can suggest the repair $\texttt{i=}k'$: `P'` $\triangleq$ `P`$[\texttt{pc} \mapsto (\texttt{i = } k')]$. More generally, if the necessary condition is $\texttt{i} \diamond k'$, with $\diamond$ a relational operator, then `i` should be initialized to a value satisfying $\texttt{i} \diamond k'$[3]. However, the initialization repair may: (i) change the behavior of the program; (ii) cause assertions not in $\delta_{\texttt{P,P'}}$ to fail in `P'`. Therefore, to verify the repair before suggesting it to the user, we analyze `P'` in background to check that no additional assertion failure is introduced by the repair, so that `P` $\sqsubseteq_a$ `P'`. In general there are many $\overline{k}'$ satisfying $\texttt{i} \diamond k'$ and `P` $\sqsubseteq_a$ `P'`. We leverage the constraint solver in (the numerical abstract domains of) `cccheck` to provide a satisfiable assignment. In most of the cases we get the most general $\overline{k}'$.

EXAMPLE 3. In Fig. 1, `Cccheck` infers the necessary condition $1 \leq$ `i`, finds the initialization `i = 0`, and suggests repairing it with `i = 1`. Similarly, in Fig. 3, `Cccheck` propagates the constraint $1 <$ `args.Length`, finds the array creation setting `args.Length` $= 1$, and suggests the repair of a larger allocation `new object[2]`.

Repairing Guards. We can use $\mathbb{B}(\texttt{e})$ to check whether a guard is too weak or even contradictory. If at a program point `pc`, $\texttt{P}(\texttt{pc}) =$ `Assume g`, *i.e.*, `g` is the guard at program point `pc` [4], and $\mathbb{B}_{\texttt{pc}}(\texttt{e}) =$ `!g`, then we can suggest to use `!g` instead of `g`, after checking that no new assertion failure is introduced. Similarly, if `g` $\triangleq$ `a <= b` and $\mathbb{B}_{\texttt{pc}}(\texttt{e}) =$ `a < b`, we can suggest a guard strengthening (after an additional run of `cccheck`). Therefore `P` $\sqsubseteq_a$ `P`$[\texttt{pc} \mapsto \texttt{Assume}(\mathbb{B}_{\texttt{pc}}(\texttt{e}))]$.

EXAMPLE 4. In the example of Fig. 4, the safety condition `c != null` contradicts the if-statement guard `c == null`. Hence it is proposed as a verified repair: all the assertions after `(*)` are unreached in `P`, and the else branch is empty. A new run of `Cccheck` is therefore not necessary. Please note that `false` may also be proposed as a guard, *i.e.* the branch of the conditional can be removed altogether.

In the example of Fig. 5, the assertion `num < callA.Length` is stronger than the (implicit else-)condition `callA.Length >= num`, and hence proposed as a repair [5].

Ensuring correct object initialization. When `e` is an assertion in a *public* method `m` and $\mathbb{B}_{\texttt{entry}}(\texttt{e})$ involves only private fields of `this` object, then $\mathbb{B}_{\texttt{entry}}(\texttt{e})$ is a necessary invariant on the object fields. In particular, if $\mathbb{B}_{\texttt{entry}}(\texttt{e})$ contains only readonly fields [6] then it should hold after the invocation of any of the constructors. Otherwise, suppose the constructor `c` does not establish $\mathbb{B}_{\texttt{entry}}(\texttt{e})$. Then the sequence of calls `c`, `m` will cause the assertion `e` to fail. We can repair it in two ways. The first way is to repair the constructor `c` so to establish $\mathbb{B}_{\texttt{entry}}(\texttt{e})$. This alternative corresponds to assuming $\mathbb{B}_{\texttt{entry}}(\texttt{e})$ to be (part of) the object invariant, and repairing the constructor to meet the invariant. We should check that the repair does not violate some other assertion (to ensure that `P` $\sqsubseteq_a$ `P'`). The second way is to deny the invocation of `m` when the object is created via `c`. This corresponds to the object invariant $\texttt{b}_\texttt{c} || B_{\texttt{entry}}(\texttt{e})$. The Boolean flag $\texttt{b}_\texttt{c}$ captures whether the class was initialized via the constructor `c`. The object invariant removes only bad runs so that: `P` $\sqsubseteq$ `P'`.

```
public class MyClass
{
 private readonly SomeObj s;

 public MyClass(SomeObj s)
 {
   Contract.Requires(s != null);

   this.s = s;
 }

 public MyClass()
 {
 }

 public int Foo()
 {
   return this.s.f;
 }
 // ...
}
```

Figure 10. An example of object initialization repair. The repair can either initialize `this.s` to a non-null value in `MyClass()` or add an object invariant to avoid the null dereference of `s` in `Foo`.

EXAMPLE 5. Let us consider the class in Fig. 10, abstracting a common pattern in system code. When `Foo` is invoked `this.s != null` should hold necessarily. Otherwise the client code:

```
x = new MyClass();
x.Foo();
```

causes a null dereference exception. To rule out bad executions we can either modify the implementation of `MyClass` or we can add a contract to specify the correct usage pattern.

The field `this.s` is private, so it cannot be made a precondition of `Foo`: it is a condition on the object fields that should be established on object creation. The first constructor, `MyClass(SomeObj)`, satisfies it. The second constructor, `MyClass()`, does not. Adding the assignment `this.s = new SomeObj()` to `MyClass()` removes the problem. An

[3] Note that k does not satisfy $\texttt{i} \diamond k'$, otherwise `cccheck` should have proven it before.

[4] As common, we assume that conditional and loop guards are represented by `Assume` statements, and control flow modelled by `gotos`.

[5] In an early stage of `Cccheck` pipeline, all such assumptions are made explicit.

[6] A readonly field can only be assigned inside constructors.

alternative is to add the object invariant

```
Contract.Invariant(this.b || this.s != null);
```

where the readonly private Boolean field is assigned `this.b = true` in `MyClass()`.

Method purity. Suppose that the necessary condition $B_{\mathtt{pc}}(\mathtt{e})$ mentions an object on the heap `o`, *e.g.*, `o.f != null`. If the instruction at `pc` is a method call and no information is provided on `m` then `Cccheck` assumes the worst for `o`, and the object gets havoced. As a consequence, the propagation of $B(\mathtt{e})$ stops at `pc`. A repair is to mark the method `m` as `[Pure]`, *i.e.*, its execution as no visible side-effects. The purity marker has no effect on the concrete semantics, but it largely improves the precision of static analyzers: `P` $\sqsubseteq$ `P'`.

6.3 Repairs Inferred From the Abstract Domains

Off-by one. The semantic facts inferred at a given program point can be used to suggest repairs. In particular, one can use the information inferred by the numerical abstract domain(s) to suggest repairs for off-by-one errors. If `cccheck` cannot prove an assertion `a < b` at program point `pc` *but* it can prove `a <= b`, then it can suggest using `a-1` instead of `a`, provided it does not introduce any new warning. In this case, `P` $\sqsubseteq_a$ `P`$[$`pc` $\mapsto$ `P(pc)`$[$`a` $\mapsto$ `a-1`$]]$.

EXAMPLE 6. In the example of Fig. 3, `cccheck` trivially finds that `1 <= args.Length = 1`, and as `0 < 1`, it suggests `0` as new array index. In the example of Fig. 5, it can prove that `num <= callA.Length`. However, it does not suggest replacing `num` with `num-1` as this may introduce a new bug in the program (a buffer underflow at the same line).

Floating point comparison. The .NET type system allows two kinds of floating point numbers `Float32` (32 bits) and `Float64` (64 bits). The .NET specification requires that floats in locals (stack locations, parameters, return values) *should* be implemented by the underlying virtual machine with the highest precision available from the hardware (*e.g.*, 80 bits registers in `x86` architectures). On the other hand, heap locations (fields, array elements, statics) *should* always match the precision of their nominal type. As a consequence, when a local float is stored into a heap location its value is truncated. The comparison of values of different bit sizes may lead to very unexpected results.

For an expression `a`$\diamond$`b` at `pc`, with $\diamond$ relational operator, if `cccheck` deduces that one of the operands has an extended precision, while the other has nominal precision, it suggests the repair `t`, containing the truncation of the extended precision value to its nominal type. Therefore, `P` $\sqsubseteq$ `P`$[$`pc` $\mapsto$ `t`$]$.

EXAMPLE 7. In Fig. 6, `cccheck` infers the parameter `d0` (of extended precision) is compared against a constant (of 64 bits), and it suggests adding the cast `(double)d0` to force the truncation and guarantee that the comparison operator checks floating point values of the same bit size. In `P'` the object invariant is hence satisfied.

Arithmetic Overflows. We introduce a new algorithm to repair arithmetic overflows that leverages the decision procedure and the numerical facts inferred by `Cccheck` abstract domains. We consider expressions in the language:

$$
\begin{array}{lcl}
\mathtt{e} & ::= & \mathtt{a} \mid \mathtt{a} \diamond \mathtt{a} \\
\mathtt{a} & ::= & k \mid \mathtt{v} \mid \mathtt{a + a} \mid \mathtt{a - a} \mid \mathtt{a \ / \ k} \\
\diamond & ::= & \mathtt{<} \mid \mathtt{<=} \mid \mathtt{>} \mid \mathtt{>=} \mid \mathtt{==} \mid \mathtt{!=} \\
k & ::= & -2^{p-1} \mid \ldots -1 \mid 0 \mid 1 \mid \ldots 2^{p-1}-1
\end{array}
$$

The rewriting rules are in Fig. 11. They immediately induce a non-deterministic memoization-based algorithm. The algorithm starts with an expression `a` which may cause an overflow for some input, and rewrites it to an expression `a'` which is provably non-overflowing. The algorithm annotates each subexpression with a tag: ? means that we do not know if the expression may overflow; ! means that the expression is provably not-overflowing (for the values in the concretization of the current abstract state). If it succeeds, the algorithm guarantees that `a'`: (i) evaluates to the same value as `a` when they are both interpreted over $\mathbb{Z}$; and, (ii) no overflow happens when evaluated on $\mathbb{Z}_p$, where $p \in \{8, 16, 32, 64 \ldots\}$ is the given integer precision.

The algorithm is incomplete by design, for performance reasons. It is an abstract interpretation of the trivial algorithm which enumerates all the equivalent expressions and then checks for non-overflowing. Next we detail the rules. A constant, a variable, and the comparison of non-overflowing expressions do not overflow. We can remove the uncertainty on an binary arithmetic expression if the underlying abstract state guarantees that the operation does not overflow. Moving the right operand of a subtraction to the right of a comparison operator removes a possible overflow. In the case of an addition, one should pay attention that `-a` does not overflow (*i.e.*, `a` may be `MinInt`). Division by a constant overflows if $k = 0$ or if `MinInt` is divided by -1. Half-sum can be written in two ways (note that the rule `(a+b)`$/2 \rightarrow$ `a`$/2 +$ `b`$/2$ is incorrect when `a`, `b` are odd quantities). We can trade an addition for a subtraction, or a subtraction for an addition if we are guaranteed that the new expression does not overflow. Finally, we allow shuffling expressions by moving them on the same side of a relational operator, and we introduce strict inequalities to remove overflows (the dual rule for `>=` is not shown in the figure). Let `P'` be such that all the overflowing expressions are replaced by the result of the algorithm above. Then `P` $\sqsubseteq_a$ `P'`.

EXAMPLE 8. In the binary search example (Fig. 7), `Cccheck` proves that all but one arithmetic expressions are not-overflowing. It infers the loop invariant $0 \leq$ `inf` $\leq$ `sup` $<$ `array.Length`. As a consequence, at `(*)`, it can apply the rule for the half-sum (the difference of two non-negative values can never overflow), and it suggest the correct expression. Similarly, in Fig. 8, `cccheck` captures the possible overflow in the addition, and it suggests using a subtraction instead.

$$\frac{}{k^? \to k^!} \quad \frac{}{\mathtt{v}^? \to \mathtt{v}^!}$$

$$\frac{ok(\mathtt{a}_1 op \mathtt{a}_2) \quad op \in \{+,-\}}{(\mathtt{a}_1^! op \mathtt{a}_2^!)^? \to (\mathtt{a}_1^! op \mathtt{a}_2^!)^!}$$

$$\frac{ok(\mathtt{-a}_2)}{((\mathtt{a}_1^! + \mathtt{a}_2^!)^? \diamond \mathtt{0})^? \to (\mathtt{a}_1^! \diamond \mathtt{-a}_2^!)^!}$$

$$\frac{}{((\mathtt{a}^! + \mathtt{b}^!)^?/\mathtt{2}^!)^? \to ((\mathtt{a}^! + ((\mathtt{b}^! - \mathtt{a}^!)^?/\mathtt{2}^!)^!)^!)^!}$$

$$\frac{ok(\mathtt{c} - \mathtt{b})}{((\mathtt{a}^! + \mathtt{b}^!)^? \diamond \mathtt{c}^!)^? \to (\mathtt{a}^! \diamond (\mathtt{c}^! - \mathtt{b}^!)^!)^!}$$

$$\frac{ok(\mathtt{a-c})}{((\mathtt{a}^! + \mathtt{b}^!)^? - \mathtt{c}^!)^? \to ((\mathtt{a}^! - \mathtt{c}^!)^! + \mathtt{b}^!)^?}$$

$$\frac{ok(\mathtt{a+b})}{((\mathtt{a}^! - \mathtt{c}^!)^! + \mathtt{b}^!)^? \to ((\mathtt{a}^! + \mathtt{b}^!)^! - \mathtt{c}^!)^?}$$

$$\frac{}{((\mathtt{a}^! + \mathtt{b}^!)^? \diamond \mathtt{c}^?)^? \to (((\mathtt{a}^! + \mathtt{b}^!)^? - \mathtt{c}^?)^? \diamond \mathtt{0})^?}$$

$$\frac{}{(\mathtt{a}_1^! \diamond \mathtt{a}_2^!)^? \to (\mathtt{a}_1^! \diamond \mathtt{a}_2^!)^!}$$

$$\frac{}{((\mathtt{a}_1^! - \mathtt{a}_2^!)^? \diamond \mathtt{0})^? \to (\mathtt{a}_1^! \diamond \mathtt{a}_2^!)^!}$$

$$\frac{k \neq 0 \wedge (\mathtt{a} \neq \mathtt{MinInt} \vee k \neq -1)}{(\mathtt{a}^!/k^!)^? \to (\mathtt{a}^!/k^!)^!}$$

$$\frac{}{((\mathtt{a}^! + \mathtt{b}^!)^?/\mathtt{2}^!)^? \to ((\mathtt{b}^! + ((\mathtt{a}^! - \mathtt{b}^!)^?/\mathtt{2}^!)^!)^!)^!}$$

$$\frac{ok(\mathtt{c} - \mathtt{a})}{((\mathtt{a}^! + \mathtt{b}^!)^? \diamond \mathtt{c}^!)^? \to (\mathtt{b}^! \diamond (\mathtt{c}^! - \mathtt{a}^!)^!)^!}$$

$$\frac{ok(\mathtt{b-c})}{((\mathtt{a}^! + \mathtt{b}^!)^? - \mathtt{c}^!)^? \to (\mathtt{a}^! + (\mathtt{b}^! - \mathtt{c}^!)^!)^?}$$

$$\frac{ok(\mathtt{a+b})}{(\mathtt{a}^! + (\mathtt{b}^! - \mathtt{c}^!)^?)^? \to ((\mathtt{a}^! + \mathtt{b}^!)^! - \mathtt{c}^!)^?}$$

$$\frac{}{((\mathtt{a}^! + 1^!)^? \mathtt{<=} \mathtt{b}^!)^? \to (\mathtt{a}^! \mathtt{<} \mathtt{b}^!)^!}$$

Figure 11. The rules used by the overflowing expression repair algorithm. The function $ok(\mathtt{a})$ uses the inferred facts to check whether the expression a (or one of its subcomponents) does not overflow.

Library	Methods	Overall Time	Asserts	Validated	Warnings	Repairs	Time	Asserts with Repairs	%
system.Windows.forms	23,338	62:00	154,863	137,513	17,350	25,501	1:27	14,617	84.2
mscorlib	22,304	38:24	113,982	103,596	10,386	16,291	0:59	7,180	69.1
system	15,187	26:55	99,907	90,824	9,083	15,618	0:47	6,477	71.3
system.data.entity	13,884	51:31	95,092	81,223	13,869	28,648	1:21	12,906	93.0
system.core	5,953	32:02	34,156	30,456	3,700	9,591	0:27	2,862	77.3
custommarshaler	215	0:11	474	433	41	31	0:00	35	85.3
Total	**80,881**	**3:31:03**	**498,474**	**444,045**	**54,429**	**95,680**	**4:51**	**44,077**	**80.9**

Figure 12. The experimental results of verified repairs on the core .NET libraries. We report the number of methods, the overall analysis time, the number of assertions, validated assertions, warnings, the number of repairs, the time it took to infer them, the number of assertions with at least one repair and the percentage of warnings with at least one repair. Time is in minutes.

7. Experience

Shipped Libraries We generated verified repairs for the *shipped* core libraries of the .NET framework (they are not yet annotated with contracts). We run Cccheck with the default checks (nonnull, bounds, arithmetic) and the precondition, postcondition and object invariant inference on. Inference propagates preconditions and object invariants to the caller, where they become new assertions. The more refined the inference algorithm, the more complex the inferred preconditions [6].

The empirical results are in Fig. 12. For each assembly, we report the total number of analyzed methods and the overall analysis time. We also report the number of assertions, validated assertions (*true*, *bottom*), and warnings (*false*, *top*). The total number of repairs is obtained *after* simplification and includes *all* of the repairs enumerated in Sec. 6. We use a simple simplification procedure to eliminate redundant repairs. In general, the simplification procedure takes as input a set of repairs R and returns a set of repairs R' such that: (i) the repairs in R and R' are equivalent; and (ii) $\#R < \#R'$. For instance, if `x > 0` and `x > 1` are inferred as repairs, we retain only the latter. The time includes the time to infer the repairs and simplify them. We also report the number of warnings for which we discovered *at least* one repair (in general several repairs are possible for the same warning) and the percentage over the total number of warnings.

Our first observation is that we can infer thousands of repairs in large libraries in a short time – the time to generate code repairs is only a tiny fraction of the overall run of Cccheck. We can propose a repair for a warning in almost 81% of the cases (overall). If the programmer decides to apply one of the repairs, the precision of the analyzer (the

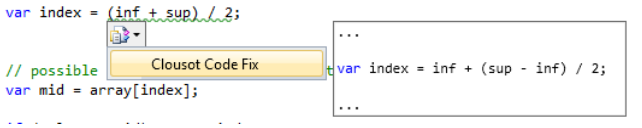

Figure 13. The code repairs in the IDE.

percentage of all asserts that `cccheck` validates) rises by almost 10% (from 89% to almost 99%). We cannot apply repairs automatically, as in general there is more than one repair possible for a given warning.

Automatic application of repairs is out-of-the scope of our work: we want to provide the programmer with various *semantically* justified choices and let her decide how to repair the code. Repairs provide a witness for the warnings, and we use them to rank the warnings. We manually inspected some of the repairs generated by `Cccheck` and discovered new bugs in shipped and very well-tested libraries. Some of those bugs are illustrated by the examples in Sec. 2.

IDE integration The goal of verified repairs is to use them interactively. We integrated `Cccheck` in Visual Studio, via the Microsoft Roslyn CTP [26]. An example of the user experience is in Fig. 13. Roslyn exposes the C# (and VB) compiler(s) as a service. Plugins register to Roslyn as code issues and code action providers. `Cccheck` runs in background while the programmer is writing the program. `Cccheck` reports the assertion violations as code issues. For those assertions it can infer a code repair, `Cccheck`, reports it as a code action. The code action is shown in a preview window. The user choses to apply the code action or not.

Verified code repairs are realistic only if we can prove that the running time of the analysis + the generation of the fixes is small enough to be used in a real time programming environment. On average `Cccheck` analyzes $6+$ methods per second and it infers 7.5 repairs per second (Fig. 12). The benchmarks above were run without any form of caching. However, to further improve the performances, `Cccheck` has a built-in cache mechanism which let the analysis run only on the methods that have been modified since the last analysis. In the past, we measured the effects of the cache mechanism to at least a $10\times$ speed-up.

Overall, we are very positive that the approach can be applied during active development: code repairs + `Cccheck` with cache are efficient enough to be used in the IDE.

8. Related Work

Automatic program repair is an active research subject in the testing community [28]. Starting from a program `P` exposing a bug, the goal is to derive `P'` such that the bug is repaired and no regression is introduced. The repair is obtained by pseudo-invariant inference [27], by genetic algorithms [19, 35], by a smart exploration of the space of repairs [1], or by instantiating some templates [34]. Our approach is different in that we do not use a *known* failing test to infer the repair (we start from a warning issued by the analyzer), we do not need to run the program to apply our technique, the repair is inferred from the semantics of the program and it is verified. In particular, the repairs we generate are property-specific, so our technique is not subject to the "randomness" in the proposed repairs of the aforementioned techniques. Pex [32] uses symbolic execution to remove inputs that inevitably lead to an error. Our fixes are more general (*e.g.*, we can fix overflowing expressions) and they soundly cope with loops – symbolic execution engines do not infer loop invariants. Starc [9] mixes dynamic and static analysis to repair data structures. We do tackle the problem of repairing data structures, but an interesting future research direction is to extend the definitions of Section 4 in that sense. Samini *et al.* [31] also combines dynamic and static analysis, but to repair PHP programs.

Eclipse [13] has a fix-it feature for repairing *syntactically* wrong programs. Whalen *et al.* [36] have a vision of an integrated *test* environment. Our vision is instead that of a *semantic* integrated environment [21], where a static analyzer runs in background, reports the errors, proposes the semantically justified repairs, and helps common programming tasks such as refactoring [7], searching, code reviewing *etc.*.

In the static analysis and verification community, recent work focused on the repairing of Boolean programs [14, 16, 30]. We are not constrained to Boolean programs (we handle arbitrary C# programs), we handle infinite state spaces, and we have a very precise yet universal notion of what a code repair is. We were able to generate repairs for the running examples of the above papers, with the exception of the concurrent examples of [16] – `Cccheck` does not analyze parallel programs. Surprisingly enough, for the running example of [14] we inferred a different repair (the initialization `x = 1`).

Other authors focused on repairing programs for specific properties. Martel [24] tackles the problem of repairing *floating* point arithmetic expressions. The goal of [24] is to infer (a good-enough approximation of) the most precise expression over machine floats equivalent to a given one. His work is a code repair as according to our Definition 2. The main differences with the algorithm in Fig. 11 is that we focus on `int`s, so we are only interested in *a* non-overflowing expression (there may be many in general), we perform online local rewriting instead of estimating the cost for the whole expression, we use all the numerical information inferred by the numerical abstract domains, not only the numerical ranges.

Vechev *et al* [33] present algorithms to introduce synchronization to remove bad interleavings. In spirit, their approach is very similar to ours, as they use abstract interpretation to verify a program, and when they fail they gave themselves the freedom to modify the program also when it cannot be proven in the abstract that it is wrong (it may be correct in the concrete). Apart from the handling of concurrency, the main difference is that we do not aim at applying

the fixes automatically, we give some semantic guarantees on the repairs (they should not remove good runs and should increase the number of validated assertions), and we do not aim for minimality.

A related research field is the localization and the explanation of bugs and warnings. Jose and Majumdar [17] present an algorithm to find the cause of a failing test. Their approach is different from ours in many ways. First, their algorithm requires the knowledge of a complete (failing) program execution – we use static analysis instead. Second, their error localization algorithm is based on finite techniques (bounded model checking, SAT, loops are handled by unrolling) so it is unlikely to cope with infinite state spaces as we do. Third, their scope is to find the origin of the bug, whereas we aim at proposing a repair for a possible error in the program. Rival [29] proposes a set of techniques to find the origin of alarms in an industrial strength static analyzer. Our verified repairs can be also used to explain the origin of the warnings of the static analyzer. Dillig, Dillig and Aiken [8] use abduction to infer the information a static analyzer is missing to carry on the correctness proof (or to prove the program is incorrect). The information is presented to the user who should validate or refuse it. We do not use abduction and we interact with the user by suggesting code changes, instead of logical formulas. To see how our technique compares to abduction, we converted the C benchmarks of [8] into C#. We skipped those relying on C-specific patterns, *e.g.*, string and pointer manipulation. We applied `cccheck` on the converted benchmarks. For most of them, `cccheck` not even shows a suggestion, as it was able to prove the correctness without additional help. For all the others, *e.g.*, the one with real errors, `cccheck` suggested meaningfull assumptions and repairs. Overall, we did not experience any advantage of the abduction technique of [8] over ours.

9. Conclusions

We envision a future in which IDEs not only report semantic errors at design time but also suggest code repairs for them [21]. This paper is a first step in that direction. We introduced a new analysis for automatic, modular and verifiable program repair. We used abstract interpretation to formalize the concepts of a verified repair (which removes bad runs while possibly increasing good runs), and the weaker notion of verified *assertion* repair. We presented a set of verified repairs, implemented in our static analyzer for .NET. The repairs are extracted from the semantic information computed by the abstract interpreter.

Experience shows that the repairs: (i) are generated fast enough that they could be computed during active development; (ii) cover almost $4/5$ of the warnings raised; and (iii) are precise enough to find new bugs for very well tested shipped libraries.

Acknowledgments

We thank Andrew Bagel, Patrick and Radhia Cousot, Manuel Fändrich, Matthieu Martel, Nikolai Tillman for the discussions. We are also gratefull to Mike Barnett who made the Roslyn integration possible.

References

[1] S. Chandra, E. Torlak, S. Barman, and R. Bodík. Angelic debugging. In *ICSE*, 2011.

[2] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.

[3] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2), 2002.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM POPL*, 1977.

[5] P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, 2011.

[6] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, 2011.

[7] P. Cousot, R. Cousot, F. Logozzo, and M. Barnett. An abstract interpretation framework for refactoring with application to extract methods with contracts. In *OOPSLA*, 2012.

[8] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, 2012.

[9] B. Elkarablieh, S. Khurshid, D. Vu, and K. S. McKinley. Starc: static analysis for efficient repair of complex data. In *OOPSLA*, 2007.

[10] M. Fähndrich. Static verification for Code Contracts. In *SAS*, 2010.

[11] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *ACM OOPSLA*, 2003.

[12] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, 2010.

[13] Eclipse Foundation. Eclipse. `http://eclipse.org`, 2011.

[14] A. Griesmayer, R. Bloem, and B. Cook. Repair of boolean programs with an application to c. In *CAV*, 2006.

[15] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10), 1969.

[16] B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *CAV*, 2005.

[17] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *PLDI*, 2011.

[18] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.

[19] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *ICSE*, 2012.

[20] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6), 1994.

[21] F. Logozzo, M. Barnett, P. Cousot, R. Cousot, and M. Fähndrich. A semantic integrated development environment. In *OOPSLA Companion*, 2012.

[22] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, 2008.

[23] F. Logozzo and M. Fähndrich. Checking compatibility of bit sizes in floating point comparison operations. In *3rd workshop on Numerical and Symbolic Abstract Domains*, ENTCS, 2011.

[24] M. Martel. Program transformation for numerical precision. In *PEPM*, 2009.

[25] B. Meyer. Applying "Design by Contract". *IEEE Computer*, 25(10):40–51, 1992.

[26] Microsoft. Roslyn CTP. `http://msdn.microsoft.com/en-us/roslyn`, 2011.

[27] J. H. Perkins, S. Kim, S. Larsen, S. P. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *ACM SOSP*, 2009.

[28] M. Pezzè, M. C. Rinard, W. Weimer, and A. Zeller. Self-repairing programs (Dagstuhl seminar 11062). *Dagstuhl Reports*, 1(2):16–29, 2011.

[29] X. Rival. Understanding the origin of alarms in astrée. In *SAS*, 2005.

[30] R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic generation of local repairs for boolean programs. In *FMCAD*, 2008.

[31] H. Samimi, M. Schäfer, S. Artzi, T. D. Millstein, F. Tip, and L. J. Hendren. Automated repair of html generation errors in php applications using string constraint solving. In *ICSE*, 2012.

[32] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *TAP*, 2008.

[33] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[34] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *ISSTA*, pages 61–72, 2010.

[35] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, 2009.

[36] M. W. Whalen, P. Godefroid, L. Mariani, A. Polini, N. Tillmann, and W. Visser. Fite: future integrated testing environment. In *FoSER*, 2010.