Counterexample-guided inductive repair of reactive contracts

Conference Paper · May 2022

DOI: 10.1145/3524482.3527650

5 authors, including:

Soha Hussein
University of Minnesota Twin Cities
11 PUBLICATIONS **84** CITATIONS
SEE PROFILE

Sanjai Rayadurgam
University of Minnesota Twin Cities
67 PUBLICATIONS **1,056** CITATIONS
SEE PROFILE

Stephen McCamant
University of Minnesota Twin Cities
75 PUBLICATIONS **4,124** CITATIONS
SEE PROFILE

Vaibhav Sharma
University of Minnesota Twin Cities
21 PUBLICATIONS **208** CITATIONS
SEE PROFILE

Counterexample-Guided Inductive Repair of Reactive Contracts

Soha Hussein*
soha@umn.edu
University of Minnesota
USA

Sanjai Rayadurgam
rsanjai@umn.edu
University of Minnesota
USA

Stephen McCamant
mccamant@cs.umn.edu
University of Minnesota
USA

Vaibhav Sharma
vaibhav@umn.edu
University of Minnesota
USA

Mats Heimdahl
heimdahl@umn.edu
University of Minnesota
USA

ABSTRACT

Executable implementations are ultimately the only dependable representations of a software component's behavior. Incorporating such a component in a rigorous model-based development of reactive systems poses challenges since a *formal contract* over its behaviors will have to be crafted for system verification. Simply hypothesizing a contract based on informal descriptions of the component is problematic: if it is too weak, we may fail in verifying valid system-level contracts; if it is too strong or simply erroneous, the system may fail in operation. Thus, establishing a valid and strong enough contract is crucially important.

In this paper, we propose to repair the invalid hypothesized contract by replacing one or more of its sub-expressions with newly composed expressions, such that the new contract holds over the implementation. To this effect, we present a novel, sound, semantically minimal, and under reasonable assumptions terminating, and complete counterexample-guided general-purpose algorithm for repairing contracts. We implemented and evaluated our technique on more than 4,000 mutants with various complexities generated from 29 valid contracts for 4 non-trivial Java reactive components. Results show a successful repair rate of 81.51%, with 20.72% of the repairs matching the manually written contracts and 60.79% of the repairs describing non-trivial valid contracts.

ACM Reference Format:
Soha Hussein, Sanjai Rayadurgam, Stephen McCamant, Vaibhav Sharma, and Mats Heimdahl. 2023. Counterexample-Guided Inductive Repair of Reactive Contracts. In *Proceedings of ACM Conference (FormaliSE'22).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3524482.3527650

*Also with Ain Shams University, Egypt
Lecturer on leave of absence
soha.hussein@cis.asu.edu.eg.

FormaliSE'22, May 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9287-7/22/05…$15.00
https://doi.org/10.1145/3524482.3527650

1 INTRODUCTION

Computer-controlled systems are typically developed by integrating multiple reactive components. *Reactive components* are implemented to indefinitely execute a top-level control loop. Each iteration of the loop is a reactive *step function* in which the latest input is consumed, the component's state is updated, and a fresh output is produced. These components are sometimes developed by third-parties and the actual component delivered may be the only authentic representation of its behavior. Specifically, they may be lacking reliable requirements (or *contracts*) defining the assumptions made on a component's operating environment and the component's guaranteed behavior. Integrating such components in rigorous software development (such as in formal model-based development [7, 41]) is challenging, as one has to hypothesize their formal contracts based on the executable component and the available informal descriptions. This approach leads to two problems. If the hypothesized contract is too weak, we may fail to verify system-level contracts and subsequently discard a perfectly good component because we could not establish a more accurate (a *tighter*) contract. If the hypothesized contract is erroneous or too strong, we may succeed with the system-level verification, but the system may fail in operation since the component does not meet its hypothesized contract. Thus, establishing a valid and tight enough contract – one that does not over-approximate the component behavior to a point where verification of system-level properties fails – is crucially important.

Integrating executable components with unknown contracts into a system architecture is, by nature, an iterative process. It starts with a hypothesized formal component contract against which the component is checked. If it does not satisfy the hypothesized contract, we may have to modify the hypothesized contract until we discover one that does hold over the component. Hopefully, this new contract will allow for verification of system-level contracts (the contract is strong enough); if not, we may have to find a different component, re-architect the system, or relax the system-level contracts to reflect the reality of the components available [42]. Either way, an approach and a tool that allows an analyst to hypothesize a component contract, check if the contract is met by the low-level code implementing the component, and, if not, have an automated support to help "repair" the contract, would be helpful. In general terms, *repairing* a contract is the process of replacing one or more sub-expressions in the contract with one or more newly composed expressions, such that the new contract can be checked to be valid over the implementation.

Existing approaches do not directly address the problem of repairing a hypothesized contract. Tools that repair unrealizable specifications [20, 24], by definition, address defective contracts that no implementation can meet. Invariant discovery tools [12, 28, 43], on the other hand, generate program invariants, typically restricted to a predefined grammar, that cannot be directly related to users' hand-written contracts.

In this paper, we present a novel sound, semantically minimal, terminating, and complete general-purpose algorithm for repairing contracts, assuming effective terminating witness-finding procedures. Also, we present its instantiation to repair contracts of reactive components in a tool named `ContractDR`. The implementation is also sound and terminating but boundedly minimal and complete.

`ContractDR`[1] uses a Counterexample-Guided Inductive Repair (CEGIR) algorithm that alternates between bounded and unbounded model-checking (both constrained by a time budget). The input is a hypothesized contract and a Java bytecode implementation of the executable reactive component. The process starts by translating the component's implementation into a dataflow representation amenable to model-checking. If the hypothesized contract was invalid with respect to the implementation, our technique attempts to repair it. To do that, we punch holes (identifying repair points) in the given contract to be repaired. Then we repeatedly find *candidate contracts*, possible repaired contracts that are not yet checked to be valid over the implementation. Once our technique finds an *initial repaired contract*, the first validated contract over the implementation, it attempts to find a tighter repaired contract. In general, our technique can find multiple *repaired contracts*, candidate contracts that were checked and found valid over the implementation. The goal of our technique is to find a *minimal repaired contract*, a repaired contract that cannot be further tightened.

To generate replacement expressions needed for the repair, we use the Sketching [36] technique. Generally, sketching is a program synthesis technique that synthesizes pieces of the partially detailed program using another implementation as a reference. Our usage for the sketch technique for repairing contracts is similar, but for synthesizing a partially detailed contract, as opposed to a partially detailed program; our process tries to synthesize partial contracts using the component's implementation as a reference.

In a previous short paper [17], we described an overview of our repair approach and evaluated classes of repairs of a single benchmark. In this paper, we present our general-purpose repair algorithm and prove that it is sound, terminating, complete, and is generating semantically minimal repairs. We also describe how we instantiated it for repairing contracts of reactive components and the limitation for the instantiation. Finally, we present the results of repairing over 4,000 hypothesized contracts, obtained from 29 valid ones, spanning four non-trivial Java implementations of reactive components. In our evaluation, we answer research questions about the performance of the underlying tool, the effect of directly repairing the faulty sub-expression, the impact of a poor hypothesized contract, and finally, the effect of the complexity of the component's implementation on the repairs. Results show that our technique has a successful repair rate of 81.51%, with 20.72% of the repairs matching the manually written contracts and a further 60.79% of the repairs describing non-trivial valid contracts.

2 MOTIVATING EXAMPLE

Consider a system level contract of an infusion pump—the Generic Patient Controlled Analgesia pump (GPCA) [27]:

Requirement 59: *"If the estimated remaining drug volume drops below the empty-threshold during infusion, the infusion shall stop."*

formalized as $(On \wedge Therapy \wedge Empty) \rightarrow (Rate = 0)$. We will consider two components in the GPCA architecture, the $Alarm_c$ component and the $Infusion_c$, which collaborate to help meet the GPCA requirement (REQ 59). The GPCA architecture allocates the responsibility of detecting exceptional conditions and setting the appropriate warning levels to the $Alarm_c$: *"If during infusion the drug volume drops below the empty-threshold, the alarm shall be set to 4.* Formally, $(On \wedge Therapy \wedge Empty) \rightarrow (Alarm = 4)$. Finally, the drug flow is managed by the $Infusion_c$: *"If the system is On, and the alarm level is 4, the infusion shall stop."*, formalized as $(On \wedge Alarm = 4) \rightarrow (Rate = 0)$.

Given the component contracts, we can verify that the system level contract (Requirement 59) holds. Now, assume that the actual $Alarm_c$ component we are planning on using has been developed by a third party. Suppose this component relied on is now obsolete and implements: $(On \wedge Therapy \wedge Empty) \rightarrow (Alarm = 5)$.

Clearly, integrating this component will (1) violate the $Alarm_c$ component contract and (2) lead to a violation of Requirement 59 since the infusion will not stop when it should. Through verification of the bytecode, we can show that the desired contract is not met; but, that is all we will know at this point. The question we will face now is, can we repair our invalid but desirable hypothesized contract $(On \wedge Therapy \wedge Empty) \rightarrow (Alarm = 4)$, to better reflect the actual behavior of the delivered bytecode? This will allow us to modify our architecture to accommodate the component and still meet our system-level requirement. Note that we are not interested in generating just any invariant or contract related to our component; it is likely that the contract we seek is closely related to our hypothesized contract. Thus, the search for a repair (a modification) of the hypothesized contract should take place in the space of repairs syntactically close to the invalid hypothesized contract.

`ContractDR` attempts repairing parts of the hypothesized contract of a single component (compositional repair of multiple contracts is left for future work). `ContractDR` uses *semantic minimality* as a guiding principle to avoid non-useful weak formulas. To see why, one possible contract repair to the assumed third-party component of the Alarm_c can be $(On \wedge Therapy \wedge Empty) \rightarrow (Alarm \leq 5)$. Though the repair is valid for the component it is not good enough for compositional verification. Generating $(On \wedge Therapy \wedge Empty) \rightarrow (Alarm = 5)$, is a more useful repair since this will allow us to better understand our component and allow us to modify the contract for the $Infusion_c$ so that we can verify Requirement 59 (or discard the third-party component and produce a new one meeting our original desired contract). For this reason, our technique attempts to find semantically minimal repairs rather than *any* repair.

3 RELATED WORK

The closest previous work in repairing of specifications addressed repairing unrealizable specifications to make them realizable [6, 9, 22, 24]. These tools make a specification realizable only by refining

assumptions, which limits the set of starting specifications that can be repaired. By comparison, our faulty specifications are usually already realizable but are not realized by the available implementation and our repairs often require changes beyond adding assumptions. The algorithms we use are more closely inspired by program repair [10, 19, 23, 25, 26]. Program repair, however, searches for modifications to improve imperative software instead of changing specifications. Our language of repair definitions is inspired by sketching [36], a programming technique that allows programmers to provide the framework and building blocks of a correct program, leaving an automated synthesis approach to search for a way of combining the pieces to create a correct final program. Sketch has also been used in program repair [16, 21], while we use it here for repairing specifications.

Invariant discovery is a technique for finding program properties such as function pre- and post-conditions and object invariants [5, 12]. For example, Daikon [12] infers likely invariants by instantiating templates over data from test executions, and uses statistical tests to exclude over-fit properties. DIG [29] uses more specialized algorithms to discover complex non-linear and array properties. PIE [32] avoids the limitation of a fixed grammar by using program synthesis and machine learning to suggest new invariants. The quality of invariants can be improved with static information [34], or by combining with automatic test-case generation [28, 30, 43]. The usage of counter-example test cases in the latter tools is analogous to our counter-example guided inductive approach. However, the goal of these tools is to generate as many correct invariants as possible from a large grammar, whereas `ContractDR` repairs a single candidate into a similar but correct specification. In the evaluation section, we empirically show that invariant discovery is insufficient for repairing contracts.

Another class of applications of invariant discovery is discovering inductive (e.g., loop) invariants to prove a given functional correctness property. Many approaches have been explored including machine learning tools [14, 35], but template-based techniques [37, 38] are the closest to our repair approach. The template enumerates a finite but large set of possibilities which must be explored efficiently. The requirements on an inductive invariant can be formulated as a kind of optimality, but the details differ from our minimality approach.

4 GENERAL-PURPOSE REPAIR ALGORITHM

Algorithm 1 finds a minimal repair, if one exists, to fix a contract sketch s for an implementation f, where in our context, a *sketch* is a partially specified contract. A *repair* is simply the values v for the *holes* (arguments) in the sketch s such that $s(v)$ is a contract for the given implementation f, i.e., all I/O behaviors exhibited by f are allowed by $s(v)$. A repair is *minimal* if no other repair can result in a stronger contract (one that disallows more I/O behaviors and is still a contract for f conforming to s). The descriptions below for the variable names may be helpful for understanding:

- A is a set of must-**a**dmit behaviors (I/O pairs).
- D is a set of may-**d**iscard behaviors (I/O pairs).
- k is a boolean indicating whether we have a **k**nown repair, i.e., we have found a repair.

Algorithm 1: General-purpose algorithm for finding repairs.

1 **discover-repair**$\Big(f : I \to O, s : H \to 2^{I \times O}, v : H\Big) : \mathbb{2} \times H$
2 $A, D, k, v' \leftarrow \emptyset, \emptyset, \neg\top, v$
3 $q \leftarrow \lambda b, u_1, u_2 \,.\, b \in s(u_1) \wedge b \notin s(u_2)$
4 **while** $\top$ **do**
5 **if** [1]$\exists ?i \in I \,.\, (i, f(i)) \notin s(v')$ **then** $A \leftarrow A \cup \{(i, f(i))\}$
6 **else if** $\exists ?d \in I \times O \,.\, q(d, v', v) \wedge k$ **then** $D \leftarrow D \cup \{d\}$
7 **else** $k, v \leftarrow \top, v'$
8 $p \leftarrow \lambda u \,.\, \forall a \in A \,.\, a \in s(u) \wedge \forall d \in D \,.\, d \notin s(u)$
9 **if** $\exists ?u \in H \,.\, p(u) \wedge (k \Rightarrow \exists x \,.\, q(x, v, u))$ **then** $v' \leftarrow u$
10 **else return** k, v

- v (when $k = \top$) and v' are the **v**alues for holes in the sketch from a known repair and a new candidate respectively.
- q is a predicate to check if u_2 **q**ualifies as a potentially tighter candidate than u_1 due to b; i.e., b is a behavior disallowed by the contract $s(u_2)$ but allowed by $s(u_1)$.
- p is a predicate to check if a candidate u precisely **p**artitions behaviors w.r.t. A and D; i.e., every must-admit behavior, but no may-discard behavior, is allowed by $s(u)$.

The algorithm is invoked with an implementation f, a sketch s for its contract and an initial repair candidate v such that $s(v)$ is the faulty contract in need of repair. It then repeatedly:

- First checks if there is an implementation behavior that is disallowed by the candidate and if so adds it to the must-admit set (line **5**).
- If otherwise, next checks if there is a behavior disallowed by the last known repair that is now allowed by the candidate and if so adds it to the may-discard set (line **6**).
- Otherwise, updates the known repair to be the candidate (line **7**).
- Finally, if a potentially tighter candidate exists—one that separates must-admit behaviors from may-discard behaviors and also disallows some behavior allowed by the known repair—makes it the candidate (line **9**).
- If no such tighter candidate exists, returns the last known repair, if any (line **10**).

Soundness is guaranteed by the fact that the control-flow will not reach line **7**, where the known repair is updated, unless the if-condition in line **5** is false, which ensures that the known repair, v when $k = \top$, will not result in any implementation behavior being disallowed by the repaired spec $s(v)$.

Minimality of the repair, if one is found, is guaranteed by the facts that (i) control-flow will not reach line **10,** where the procedure returns, unless the if-condition in line **9** is false, ensuring that there is no tighter candidate separating the must-accept (A) from the may-discard (D) behaviors, and (ii) any known repair, v when $k = \top$, separates A from D, as explained below.

Initially the candidate v' trivially separates A from D per line **2**. The definition of p in line **8** and the if-condition in line **9** ensure that any candidate v' that is ever considered separates A from D at the time of consideration in line **9**. Therefore, when the loop begins

[1]The question mark after the existential quantifier is to indicate that a witness—value for the quantified variable that makes the formula true—must be obtained, if one exists. The use of the quantified variable in the "true" branch is thus justifiable.

a new iteration at line **5**, the candidate v' separates A from D. If the condition in line **5** is true (i.e., candidate is not a repair) then the set A is updated by adding a behavior exhibited by f and so if there is a known repair, v when $k = \top$, it will remain a separator of A from D. Otherwise, if the condition in line **6** is true (i.e., candidate is not a tighter repair), then the set D is updated by adding a behavior that is disallowed by the known repair, v when $k = \top$, which will again remain a separator of A from D. If neither condition is true, then the candidate, which becomes the new known repair in line **7**, allows all behaviors of f which includes all of A, and is tighter than the previous known repair and thus disallows any previously disallowed behaviors which includes all of D.

Termination is guaranteed **if** there are effective witness-finding procedures for checking the existentially quantified formulas in the three if-conditions **and if** the number of s-equivalent partitions of the set of all I/O behaviors is finite. Two behaviors are considered *s-equivalent*, if for every v, either both are allowed or both are disallowed by $s(v)$. In particular, a finite H is sufficient to guarantee finiteness of s-equivalent partitioning since no partition will be considered more than once.

Completeness is guaranteed **if** the conditions for termination hold since, if there is a repair u, then the procedure will not return until a repair has been found, because till then $k \neq \top$ and the may-discard set D remains empty, ensuring that the repair u satisfies the existentially quantified formula in line **9** which will prevent execution of the return statement on line **10**.

4.1 Safety Properties of Dataflow Programs

The above algorithm can be instantiated for combinations of contract and implementation logics for which witness-finding procedures to answer the existence-queries in the if-conditions are possible. Of interest in the present work are reactive control systems for which implementations can be modeled as synchronous dataflow programs and contracts as safety properties which must hold at every logical time-step of the system's execution.

A synchronous dataflow program is a (dependently-typed[2]) function $\Delta : \forall n \geq 0 \,.\, \Sigma^n \rightarrow \Theta^n$, that, given a sequence[3] of stimuli $\sigma_1 \ldots \sigma_n$ as input, computes a sequence of responses $\theta_1 \ldots \theta_n$ as output *incrementally*, i.e., the set of all sequences of stimulus-response pairs resulting from its computation,
$\Pi^* \triangleq \bigcup_{n \geq 0} \{(\sigma_1, \theta_1) \ldots (\sigma_n, \theta_n) \in (\Sigma \times \Theta)^n \mid \Delta(\sigma_1 \ldots \sigma_n) = \theta_1 \ldots \theta_n\}$
is *prefix-closed*. Such a program is in effect a state-transition system often expressed as a step function: $\delta : \Theta \times \Sigma \rightarrow \Theta$ along with a pre-response $\theta_0 \in \Theta$. The program's response at each step is determined by the previous response[4] and the stimulus at that step: $\forall i > 0 \,.\, \theta_i = \delta(\theta_{i-1}, \sigma_i)$.

Safety properties for such programs are predicates over response-stimulus-response triples, $\phi : \Theta \times \Sigma \times \Theta \rightarrow \mathbb{2}$. The program satisfies the property, i.e., $\Delta \models \phi$, if each *contiguous* such triple in every computation of the program satisfies the predicate, i.e., $\forall \overrightarrow{\pi} \in \Pi^* \,.\, \overrightarrow{\phi}(\overrightarrow{\pi})$, where, for a $\overrightarrow{\pi} = (\sigma_1, \theta_1) \ldots (\sigma_n, \theta_n)$, we say, $\overrightarrow{\phi}(\overrightarrow{\pi}) =$ $\phi(\theta_0, \sigma_1, \theta_1) \wedge \ldots \wedge \phi(\theta_{n-1}, \sigma_n, \theta_n)$. Though the safety property is simply a one-step predicate, the step function can capture notions of past-time linear temporal logic operators and thus it is relatively easy, using a library of standard definitions, to cast interesting temporal behaviors as properties. Lustre [3] is an example of a language for representing such data-flow programs.

Conceptually, the repair algorithm for dataflow programs takes as the implementation $f = \Delta$, the input type $I = \Sigma^*$, the set of all finite length sequences of stimuli, the output type $O = \Theta^*$, the set of all finite length sequences of responses, and the contract in need of repair $s(v) = \overrightarrow{\phi} \circ zip$.[5] Given in concrete terms, however, are the function $\delta : \Theta \times \Sigma \rightarrow \Theta$ along with θ_0 and a *potentially faulty* predicate ϕ as the contract to be repaired. Thus, techniques employed for checking the existence-queries in the repair algorithm must be able to deduce the appropriate n-step consequences from the 1-step representations. We use *safety model-checking*[7, 41], which, given (δ, θ_0, ϕ), answers "does $\Delta \models \phi$ hold?" with either a $\top$ or a $\overrightarrow{\pi} = (\sigma_1, \theta_1) \ldots (\sigma_n, \theta_n)$ such that $\neg\overrightarrow{\phi}(\overrightarrow{\pi})$ holds. Sometimes it may be only possible/necessary to do a bounded query, "is $\Delta \models \phi$ not violated within N steps?". With some trade-offs, such *bounded model-checking* can also be used as discussed in the sequel.

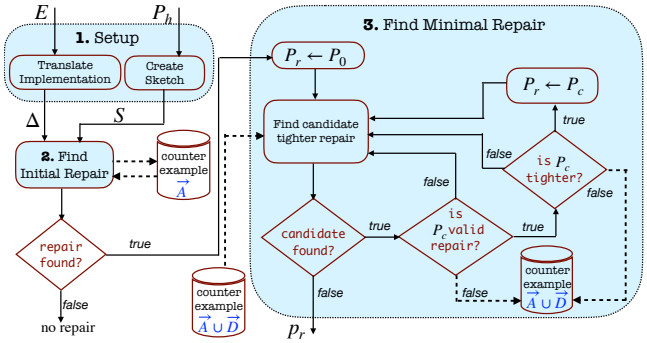

Figure 1: Overview - Focusing on Find Minimal Repair. E, S, Δ, are the executable, sketch, and logical translation of the implementation. P_h, P_0, P_c and P_r are the hypothesized, initial, candidate, and repaired contracts.

5 REPAIRING REACTIVE CONTRACTS

We now present how Alg. 1 is realized for repairing invalid reactive contracts over corresponding implementations. Fig. 1 shows an overview of the repair process, which consists of three main steps. The first step is the *setup* in which the executable implementation E is translated into its dataflow program Δ, and in which a sketch S is created by marking dubious expressions in P_h. A *dubious expression* is the expression to be replaced by a newly synthesized expression. We use enumeration to mark all possible dubious expressions. The second step is the *Find Initial Repair* step, in which both the sketch and the counterexamples are used to find the first valid repair. The counterexamples in this step correspond to must-admit behaviors more specifically; they are pairs of sequences of inputs and their matching outputs ($\overrightarrow{A}$), such that $\overrightarrow{A} : \Sigma^* \times \Theta^*$. If finding a repair

[2] The output type is a finite sequence whose length is determined by the input sequence.
[3] Separators between elements of a sequence are elided when there is no ambiguity.
[4] It is typical to distinguish the state (hidden) and output (visible) parts of the response, and take only the state part as the step function's first argument, but this distinction is not relevant here.

[5] The contract $s(v)$ takes an I/O pair (separate sequences of stimuli and responses) which must be stitched together into one sequence of stimulus-response pairs for $\overrightarrow{\phi}$.

was unsuccessful, the process terminates with no repair; otherwise, the third step, Find Minimal Repair, follows.

In the *Find Minimal Repair* step, the process tries to strengthen the initial repaired contract P_0 by finding a minimal repaired contract. To do that, we start by initializing the last known repaired contract P_r to the initial repaired contract P_0. Note that finding P_0 corresponds to having the last known repair boolean (k) (in Alg. 1) set to true. If no new candidate contract P_c was found, then the process terminates with P_r being the minimal repaired contract; otherwise, the process checks if the candidate contract P_c is matching the implementation and is tighter than P_r. If both checks are successful, then a new P_c becomes the new repaired contract; otherwise, the set of counterexamples are updated. Note that the set of counterexamples, in this case, contains both the must-admit and the may-discard sets, though again over pairs of sequences, i.e., $\overrightarrow{A} \cup \overrightarrow{D}$. The process is repeated until no other repaired contract can be found, in which case the process returns the last repaired contract P_r as the minimal repaired contract.

5.1 Setup

5.1.1 Translate the Implementation. We translate the bytecode implementation of the component into a dataflow program function Δ, which operates over sequences of stimuli and response[6]. This translation allows us to use the implementation semantics as a *reference* to repair the contract. In general, such a reactive component indefinitely executes a top-level control loop with each iteration representing a *reactive step* in which the latest stimuli is consumed, the internal state is updated, and a fresh response is produced.

The translation is done in three main steps. First, we summarize E as a Gated Single Assignment form (GSA)[31, 40], an extension of Single Static Assignment. In GSA, a logical if-then-else is used to represent different values that a variable can take along different control flow paths.

Second, from the GSA summarization we identify, (a) *free input* variables which are variables that are used but are never defined in the summarization, i.e., method input variables, (b) *output* variables, these are variables that are written to but are never used in the summarization, i.e., return variables, and finally *state input* and *state output* variables which corresponds to the first *use* and the last *def* of state variables respectively.

Finally, we create a dataflow step function (δ) where the free and state-input variables are inputs to the function, state-output, and output variables are outputs of the function. The GSA summary is the body of the function. At this point, we have created the logical representation of step functional (δ) for E, which has no stateful behavior. To recapture stateful behavior and thereby creating the dataflow program Δ, we let state-outputs flow in as state-inputs for the subsequent invocations of the step function. The initial values of variables constitute the values for the pre-response (θ_0).

5.1.2 Create a Sketch. Here we transform a hypothesized contract into a sketch. This requires: (1) identifying dubious expression in the hypothesized contract, signifying the places where new synthesized expressions are desired, and (2) defining the family of expressions that are allowed as a replacement. Thus we extend Lustre with two constructs: a *repair expression* and a *repair generator*. The former identifies which sub-parts of P we want to replace, and the latter defines the family, i.e., the grammar, of the replacement expressions.

-**Repair Expression**: is of the form `repair(`e_1`, `e_2`)`. Here, e_1 is the *dubious expression*, the location of the hole in the sketch, and e_2 is the call to a repair generator that describes the family of replacement expressions. The repair generator can reference e_1 in generating a suitable family of repair expressions, such as generating a family of expressions with a similar size as the dubious expression.

For example, the default behavior of our implementation is to try all possible repair locations for the dubious expression, among the produced attempts. One possible repair attempt for the hypothesized Alarm_c contract of REQ59 in Sec.2 can be: $(On \wedge Therapy \wedge Empty) \rightarrow \textbf{repair}((Alarm = 4), \texttt{logical\_gen}(Alarm))$.
This repair attempt tries to find a replacement for the consequent part of the implication, i.e., $Alarm = 4$. In this example, $Alarm = 4$ is e_1, the dubious expression, and `logical_gen` is e_2, the call to the repair generator `logical_gen`, which is used to constrain the family of possible replacement expressions to standard logical operations.

-**Repair Generator**: this defines the family, i.e., the grammar, of the replacement expressions. To do that we extend the definition of a node in Lustre as follows:

$$generator\_name(\overrightarrow{e})[\overrightarrow{z}] \textbf{ returns } out : bool\{body\}$$

$\overrightarrow{e}$ defines the *repair terms*, the leaves of any generated expression, while $\overrightarrow{z}$ are holes for the sketch. The boolean return variable (**out**) indicates value of the expression. For example, below is a fragment of the `logical_gen` repair generator:

```
logical_gen(i:int) [z1:int; z2:int] returns out:bool;
out = if (z1 = 0) then i < z2 else if (z1 = 1) then i > z2
    else if (z1 = 2) then i = z2 ....
```

Using the solver's valuations for z_1 and z_2, if any, `ContractDR` partially evaluates the repair generator to obtain a new replacement expression.

5.2 Repair Process

We use a CEGIR approach to repair an invalid hypothesized contract. As described above, the repair process for reactive contracts is realized using two key algorithms. The first (Alg. 2) attempts to validate the hypothesized contract, and if invalid, it repairs it by synthesizing a new expression for the dubious expression. The second (Alg. 3) attempts to find a minimal repaired contract. That is, it tries to tighten the initial repaired contract, but as other repaired contracts are found, the algorithm repeatedly attempts to tighten them until a minimal repaired contract is reached.

Both algorithms use `VALID?` and `SAT?` queries that are encoded as unbounded and bounded model checking queries, respectively.

The `VALID?` query returns a pair: a boolean and a counterexample if the `VALID?` query was falsified. The counterexample is an assignment to the free variables describing the sequence of stimuli and response pairs that falsify the validity query. Similarly, the `SAT?` query returns a pair: a boolean and a valuation v if the `SAT?` query

[6]To facilitate representation, from this point on, we leave the explicit definition of the length of the sequence (n) out of the domain and the co-domain of Δ as well as the inputs and outputs of the Δ. We just use (*) over domains and ($\longrightarrow$) over variables to represent sequences of any length, though we only consider response sequences that have the same length as the corresponding stimuli sequences as per the definition of Δ in Sec. 4

Algorithm 2: Find an *Initial* Repaired Property, where Σ^*, Θ^* and H are the domain of sequences of inputs, sequences of outputs and holes respectively.

input: Program $\Delta : \Sigma^* \rightarrow \Theta^*$
input: Sketch $S : H \rightarrow \Sigma^* \rightarrow \Theta^* \rightarrow \mathbb{B}$,
input: Val $v : H$
1 equiv $\leftarrow \neg\top$; $\quad \overrightarrow{A} \leftarrow \{\} : \Sigma^* \times \Theta^*$;
2 $P_h \leftarrow S(v)$; $\quad P_c \leftarrow P_h$;
3 **while not** *timeout* **do**
4 $\quad (equiv, (\overrightarrow{\sigma}, \overrightarrow{\theta})) \leftarrow \texttt{VALID?}(\forall \overrightarrow{\sigma} \in \Sigma^*, \overrightarrow{\theta} \in \Theta^*.(\overrightarrow{\theta} = \Delta(\overrightarrow{\sigma})) \implies P_c(\overrightarrow{\sigma})(\overrightarrow{\theta}))$;
5 $\quad$ **if** *(equiv)* **then**
6 $\quad\quad P_0 \leftarrow P_c$; $\quad P_{min} \leftarrow$ find_minimal(Δ,S,P_0, $\overrightarrow{A}$);
7 $\quad\quad$ return P_{min};
8 $\quad$ **else**
9 $\quad\quad \overrightarrow{A} \leftarrow \overrightarrow{A} \cup \{(\overrightarrow{\sigma}, \overrightarrow{\theta})\}$;
10 $\quad\quad$ (candidateFound,v') $\leftarrow$ $\texttt{SAT?}(\exists v' \in H. \bigwedge_{(\overrightarrow{\sigma_x}, \overrightarrow{\theta_x}) \in \overrightarrow{A}} S(v')(\overrightarrow{\sigma_x})(\overrightarrow{\theta_x}))$;
11 $\quad\quad$ **if not** *candidateFound* **then**
12 $\quad\quad\quad$ fail; *//unable to repair*
13 $\quad\quad P_c \leftarrow S(v')$;

was satisfiable, where the satisfying assignment for v gives the values for the holes that allow the sketch to separate the must-admit, and the may-discard behaviors. Note that, since we are using a verification engine tool to check the SAT? query, the actual formula is, in fact, a negation of a validity query.

5.2.1 Finding an Initial *Repair.* Alg. 2 attempts to find an initial repair for the hypothesized contract P_h. The algorithm instantiates part of the general-purpose Alg. 1, when k is false, to operate over the synchronous dataflow program Δ, of a pair of sequences of inputs, and sequences of outputs as opposed to a function f and a pair of input and outputs. Also, the sketch S instantiates the abstract sketch s to operate over stimuli and responses sequences. Its result is a boolean specifying whether a specific instantiation (of holes, stimuli, and responses) satisfies the desired sketch or not. Also, here, a contract P, expressing a safety property, is instantiated to operate over sequences of inputs and sequences of outputs.

Initially, we initialize the set of the must-admit sequences of behaviors $\overrightarrow{A}$ (line **1**), and instantiate the hypothesized contract P_h with the input values v, which becomes our candidate contract Pc (line **2**). The algorithm alternates between an unbounded verification query (line **4**) and a bounded synthesis query (line **10**). The unbounded verification checks whether the candidate safety contract P_c holds for all sequences of stimuli and their corresponding responses generated by the dataflow program. If it did, then P_c becomes the initial repaired contract P_0, and the find minimal algorithm is invoked to attempt to tighten it further (line **6**).

If the VALID? query was falsified, then a counterexample pair $(\overrightarrow{\sigma}, \overrightarrow{\theta})$ is generated and is added to the must-admit set $\overrightarrow{A}$ (line **9**). Then, the bounded synthesis query checks whether there is some value for the holes (v'), such that the sketch S is satisfied

Algorithm 3: Find Minimal Repaired Property, with dataflow program Δ, sketch S, and property P. Σ^*, Θ^* and H are the domain of sequences of inputs, sequences of outputs and holes respectively.

input: Program $\Delta : \Sigma^* \rightarrow \Theta^*$
input: Sketch $S : H \rightarrow \Sigma^* \rightarrow \Theta^* \rightarrow \mathbb{B}$
input: Initial Repaired Property $P_0 : \Sigma^* \rightarrow \Theta^* \rightarrow \mathbb{B}$
input: Must Admit $\overrightarrow{A} : \Sigma^* \times \Theta^*$
1 $\overrightarrow{D} \leftarrow \{\}$; $\quad P_r \leftarrow P_0$;
2 **while not** *timeout* **do**
3 $\quad$ (candidateFound,v) $\leftarrow$ $\texttt{SAT?}(\exists v \in H, \overrightarrow{\sigma} \in \overline{\Sigma^*}, \overrightarrow{\theta} \in \overline{\Theta^*}. \bigwedge_{\overrightarrow{\sigma_x}, \overrightarrow{\theta_x} \in \overrightarrow{A}} S(v)(\overrightarrow{\sigma_x})(\overrightarrow{\theta_x}) \wedge \bigwedge_{\overrightarrow{\sigma_y}, \overrightarrow{\theta}_y \in \overrightarrow{D}} \neg S(v)(\overrightarrow{\sigma_y})(\overrightarrow{\theta}_y) \wedge \neg S(v)(\overrightarrow{\sigma})(\overrightarrow{\theta}) \wedge P_r(\overrightarrow{\sigma})(\overrightarrow{\theta}))$;
4 $\quad$ **if not** *candidateFound* **then**
5 $\quad\quad$ return P_r;
6 $\quad P_c \leftarrow S(v)$;
7 $\quad$ (isTighter,$(\overrightarrow{\sigma_x}, \overrightarrow{\theta_x})$) $\leftarrow$ $\texttt{VALID?}(\forall \overrightarrow{\sigma_x} \in \Sigma^*, \overrightarrow{\theta_x} \in \Theta^*.P_c(\overrightarrow{\sigma_x})(\overrightarrow{\theta_x}) \implies P_r(\overrightarrow{\sigma_x})(\overrightarrow{\theta_x}))$;
8 $\quad$ (isMatching,$(\overrightarrow{\sigma_y}, \overrightarrow{\theta_y})$) $\leftarrow$ $\texttt{VALID?}(\forall \overrightarrow{\sigma_y} \in \Sigma, \overrightarrow{\theta_y} \in \Theta^*.(\overrightarrow{\theta_y} = \Delta(\overrightarrow{\sigma_y})) \implies P_c(\overrightarrow{\sigma_y})(\overrightarrow{\theta_y}))$;
9 $\quad$ **if** *(isTighter and isMatching)* **then**
10 $\quad\quad P_r \leftarrow P_c$;
11 $\quad$ **else**
12 $\quad\quad$ **if not** *isMatching* **then**
13 $\quad\quad\quad \overrightarrow{A} \leftarrow \overrightarrow{A} \cup \{(\overrightarrow{\sigma_y}, \overrightarrow{\theta_y})\}$;
14 $\quad\quad$ **else**
15 $\quad\quad\quad \overrightarrow{D} \leftarrow \overrightarrow{D} \cup \{(\overrightarrow{\sigma_x}, \overrightarrow{\theta_x})\}$;

for all pairs of stimuli and responses in $\overrightarrow{A}$. If the SAT? query is true, then we have a new candidate (line **13**); otherwise, the process terminates as no candidate is available (line **12**).

5.2.2 Finding a Minimal Repair. We impose three conditions while searching for a repaired contract P_c that is tighter than the last repaired contract P_r: (1) P_c must pass all previously collected counterexamples, (2) P_c must be semantically tighter than the last known repaired contract P_r, and finally (3) for all inputs, the candidate contract P_c is valid on dataflow program Δ.

In Alg. 3, the candidate repair is synthesized in line **3**. Now observe that this SAT? query is different than the SAT? query on line 10 in Alg. 2. Besides the extra constraint to ensure that all counterexamples in $\overrightarrow{D}$ (the second conjunct) cannot be satisfied, the rest of the formula requires that the new candidate has a difference on one stimulus and response pair $(\overrightarrow{\sigma}, \overrightarrow{\theta})$. This *can* indicate that the resulting candidate contract is tighter than the last known repaired contract. In other words, it is a necessary condition but not sufficient to guarantee tightness of the candidate contract. Also, note that both $\overrightarrow{\sigma}$, and $\overrightarrow{\theta}$ are elements of a bounded sequences of stimuli $\overline{\Sigma^*}$, and the bounded sequences of responses $\overline{\Theta^*}$. For practical consideration we bound the lengths of input and output

sequences. This bound ensures that the search for a new candidate has an upper limit over the length of sequences of the stimuli and subsequently the responses length. If no candidate can be found, then the last repaired contract is returned (line **5**).

If a candidate is found, we check that it is both tighter than the last known repair (line **7**) and is matching the dataflow program (line **8**). If it is, then a new repair is found (line **10**), otherwise, we update the corresponding set of counterexamples (lines **13,14**).

5.2.3 Limitations. Soundness of both Alg. 2, and Alg. 3 carries over from the general-purpose Alg. 1. Termination of both algorithms is enforced using timeouts if the search for repairs is not complete within a time budget. However, repairs generated by the instantiated algorithms are only boundedly minimal. This results from the length bound that is imposed on the set of stimuli ($\overline{\Sigma^*}$), and the set of responses $\overline{\Theta^*}$, within which the search for a tighter candidate occurs (the third and fourth conjuncts in line **3**) in Alg. 3. This means that a possible tighter repair candidate may be disregarded by our Alg. 3 because it can only be found using a longer sequence of stimuli-response that is larger than the imposed bound.

Also, as the underlying verification techniques are incomplete, we may fail to find a repair even if one exists. In our experience this accounts for 17% of the attempted repair problems. Note that, currently `ContractDR` does not incorporate loop summarization techniques[15, 39] when extracting the dataflow program and thus does not handle implementations that have loops beyond the top-level control loop. Finally, the expressiveness of the repair generators can affect the completeness and the minimality of the generated repairs. More precisely `ContractDR` will not produce a repair that exists outside the family of expressions used in the repair.

6 EVALUATION

We implemented the above technique in a tool named `ContractDR`[1]. `ContractDR`'s input is a hypothesized contract and a Java bytecode implementation of the executable reactive component. `ContractDR` creates a dataflow program representing the bytecode implementation using the summarization in the path-merging symbolic execution tool called Java Ranger (JR) [18]. JR's summary is composed in GSA format, which `ContractDR` utilizes to create the corresponding dataflow program in Lustre syntax [3]. `ContractDR` uses JKind[13], an open-source industrial-strength infinite-state model-checker for safety properties for models expressed in the synchronous data-flow language Lustre, for synthesizing and verifying repaired contracts.

We answer the following research questions:

RQ1: What is the performance of `ContractDR`*?*
RQ2: What is the effect of the dubious location?
RQ3: What is the effect of a poor hypothesized contract?
RQ4: What is the effect of the implementation complexity?
RQ5: Can dynamically-inferred invariants be used as repairs?

- Benchmarks: We used 4 Java programs Tbl. 1. Two of the benchmarks, WBS and TCAS, are widely used in the research community for evaluating verification and testing tools. The other two benchmarks, Alarm_c and Infusion_c, are from the domain of formal specification.[7] We manually translated the latter's Simulink autogenerated C code to Java and generated their bytecode versions. WBS, TCAS, and Infusion_c have only linear computation, but Alarm_c involves non-linear computation. The expected difficulty increases top to down correlated with the size of the benchmark. We obtained contracts from the assert statements in WBS and TCAS, and the requirements of the GPCA model. Only those contracts that were either proved to be true, or could not be falsified in 30 minutes, using JR, were taken as valid (total of 29 contracts).

Table 1: Benchmarks

	loc	prop.#
WBS	265	2
TCAS	300	3
Infusion_c	1,022	12
Alarm_c	1,722	10
Total	3,309	29

Table 2: General Performance of `ContractDR`.

total attempts #	4060
already matching attempts #	906
no synth attempts #	698
repairable attempts #	2456
repaired attempts #(%)	**2002 (81.51%)**
match repairs %	**20.72%**
other. relevant repairs %	**60.79%**
repaired minimal reached %	80.2%
repaired minimal SAT TO %	3.6%
repaired minimal VALID TO %	1.1%
repaired minimal cand. reached %	12.4%
repaired attempts TO %	2.5%
repaired median / avg exec. (s)	37.9 / 324.7
repaired median / avg SAT (s)	10.9 / 260.3
repaired median / avg VALID (s)	21.9 / 62.3
unrepaired # (%)	**454 (18.49%)**
unrepaired initial SAT TO%	8.1%
unrepaired Initial VALID TO %	26.4%
unrepaired initial candidates reached %	3.7%
unrepaired minimal reached %	18.5%
unrepaired minimal SAT TO %	4.0%
unrepaired minimal VALID TO %	21.4%
unrepaired minimal cand. reached %	17.8%
unrepaired median / avg exec. (s)	601.2 / 384.6
unrepaired median / avg SAT (s)	4.8 / 39.4
unrepaired median / avg VALID (s)	298.4 / 344.4

-Mutation Generation: We introduced one or two faults into contracts to generate a 1-fault mutant and a 2-fault mutant. This allows us to measure the effect of repairing poorer hypothesized contract on the repair. We used specification mutation operators [8]: Logical Operator Replacement, replace a logical operator with another logical operator, and Relational Operator Replacement, replace a relational operator with another relational operator.

-Repair Attempts: A *repair attempt* for `ContractDR` describes a particular location in the contract to be the dubious expression. By default, `ContractDR` enumerates all possible sub-expressions, as the dubious expression and attempts to repair them one by one. `ContractDR` stops when it has tried all of them. To diversify the mutation problems given to `ContractDR` to repair, instead of picking a small number of mutants and allowing `ContractDR` to enumerate all possible repair attempts, we used single runs of unrelated repair attempts. This allowed us to run 4,060 unrelated repair attempts, with more coverage of mutations/ faults.

[7] Generic Infusion Pump Research Project at https://rtg.cis.upenn.edu/gip/

To measure the effect of the location of the dubious expression, we marked repair attempts either as *inclusive* or *not inclusive* depending on whether their dubious expression includes all faults or not. Ideally, ContractDR should be able to provide repairs for the inclusive, as we know that a repair exists in this location. On the other hand, there is no such a guarantee for the not inclusive classes: it only sometimes happens that a repair in one area can compensate for a fault in another. We enforced an equal number of attempts between the inclusive and non-inclusive attempts.

-**Repair Classes:** Repairs can either be: a *matching* repair, that results in a contract that is either equivalent or tighter than the original unmutated contract, or a *relevant* repair, that results in a contract that has no logical implication relationship with the original unmutated contract, yet it describes non-trivial valid information about the component.

- **Integer Ranges:** We constrained integer ranges for synthesis by statically analyzing Java bytecode to find which constants are used with a given variable. In few remaining cases we manually identified range information that was present in the Simulink model but not in the generated code.

- **Repair Generators:** ContractDR automatically composes logical repair generators. These generators contain standard logical operations, integer comparison operators, and integer constants. Their general structure is a binary tree of logical operators, with the leaves having a boolean type. The maximum depth of the repair is one larger than the size of the dubious expression. This allows generation of new replacement expressions that are richer but not too large relative to the dubious expression.

- **Evaluation Setup:** We ran ContractDR on a sample of 70 repair attempts per contract, half inclusive-half non-inclusive of faults. We have repeated this experiment twice, one time for 1-fault mutants and another for 2-fault mutants. We used a timeout of 10 minutes for each JKind query with at most 5 iterations for finding an initial repair (Alg.2), and 30 iterations for finding a minimal repair (Alg. 3). We used Z3 [4] as the back-end SMT solver and imposed a 1-hour overall timeout per repair attempt. We used the default engines of JKind for the VALID? queries and turned off all except BMC for the SAT? queries. The maximum steps for BMC were limited to three times the maximum length of any test case to ensure boundness over the SAT? query in Alg. 3. The experiment machine ran Ubuntu 16.04.6 on a 3.6 GHz Intel Core i7-7700 CPU processor with 32 GB RAM. We used a 2GB Java heap size.

RQ1: What is the performance of ContractDR?

Tbl. 2 shows the overall performance of ContractDR. The first section describes: the total number of attempts (attempts#), the number of repair attempts of already valid contracts (already matching attempts#), i.e., the attempts whose corresponding mutated contract is not violating the component, the number of no synthesis attempts (no syn attempts#), i.e., the attempts that cannot be repaired to non-trivial repairs using the automatically composed logical repair generators, and the selected dubious location, and finally the (repairable attempts#), i.e., the attempts that ContractDR is expected to repair since they are violating the component and there exists a non-trivial repair for them. Note that the two groups of attempts: matching attempts and no synth attempts, are identified and reported by ContractDR. We excluded these classes (matching attempts and no syn attempts) from the rest of the statistics since we were interested in evaluating ContractDR only when a valid, non-trivial repair is possible.

Table 3: Match and Relevant Repairs. Times in seconds.

	match		**relevant**	
repairable attempts#	2,456			
attempts# (%)	509 (20.7%)		1,493 (60.8%)	
median/avg exec.	28.4/77.9		49.7/408.6	
median/avg SAT	8/44.5		15.4/333.7	
median/avg VALID	18.5/32.6		26.4/72.4	
	inclusive		**non-inclusive**	
	match	**relevant**	**match**	**relevant**
repaired attempts#	443	882	66	611
median/avg exec.	28.8/79.2	155.7/647.2	18.6/68.9	20.291/64.3
median/avg SAT	8.0/47.71	40.892/552.2	5.7/22.6	6.0/18.3
median/avg VALID	19/30.7	46.5/91	12.2/45.5	12.7/45.5
	1-fault		**2-fault**	
	match	**relevant**	**match**	**relevant**
repaired attempts #	303	629	206	864
median/avg exec.	21.5/63.3	63.3/491	34.8/99.2	48.9/335
median/avg SAT	6.2/39	14.41/418.1	10.2/52.5	16.2/277
median/avg VALID	13.686/23.7	28.0/70	22.2/45.8	27/76

Table 4: Performance per Benchmark. Times in seconds.

	WBS	**TCAS**	**Infusion$_c$**	**Alarm$_c$**
attempts#	280	420	1,960	1,400
matching #	73	156	479	198
no synth attempts#	63	29	309	297
repairable attempts	144	235	1172	905
repaired attempts %	**97%**	**97%**	**89%**	**66%**
minimal reached %	83%	74%	80%	81%
minimal SAT / VALID TO %	3% / 0%	17% /0%	3% /0%	1% / 4%
minimal VALID TO %	0%	0%	0%	4%
minimal cand. bound %	0%	6%	15%	13%
1-hour TO %	7%	3%	2%	2%
unrepaired %	**3%**	**3%**	**11%**	**34%**
initial SAT / VALID TO%	0% / 0%	0% /0%	25% / 0%	1% /39%
initial VALID TO %	0%	0%	0%	39%
initial cand. bound %	0%	83%	7%	1%
median / avg exec (s)	41.9 / 312	22.8 / 562	29.4 / 212	126.6 / 367
median / avg SAT (s)	24.3 / 271	12.1 / 535	9.8 / 182	8 / 95
median / avg VALID (s)	12.3 / 39	10.7 / 25	18.2 / 28	90.9 / 270

Generally, ContractDR has a successful repair rate of 81.51%, with 20.72% matching repair (a repair that is equivalent or tighter than the unmutated contract), and 60.79% relevant repairs.

- *On the performance of repaired attempts*: in the middle section of Tbl. 2, the most dominant reason for termination is reaching the minimal contract (repaired minimal reached 80.2%). Also, timing out due to generating too many invalid minimal candidates (minimal candidates reached 12.4%), are more likely to be the second cause of termination. Other termination reasons, such as query timeout (minimal SAT? TO, minimal VALID? TO), or repair attempt timeout (attempts TO), are less likely to occur.

Also, most of the attempts have relatively low execution or query times (median execution, SAT, and VALID). However, their corresponding averages are much larger; this indicates that some outliers of repair attempts take too long to terminate. Finally, the synthesis step, on average, takes most of the execution time for repair

attempts, suggesting that optimization of this query can boost performance of repaired attempts.
- *On the performance of unrepaired attempts*: the last part in Tbl. 2 shows that unrepaired attempts account for 18.49% of all the repairable attempts. Termination for this category is dominated by the initial `VALID?` query timing out, followed by the inability to synthesize a tighter contract (minimal reached 18.5%) and finally, timing out due to attempting too many candidates (minimal candidate reached 17.8%). Other less frequent terminating factors include termination due to timing out of the initial or the minimal SAT query (Initial `SAT?` TO), and minimal `SAT?` TO, reaching the maximum number of invalid initial candidates (Alg. 2) (initial cand. bound). Finally, observe that `VALID?` is more expensive than `SAT?`, indicating the difficulty of proving or disproving unrepaired contracts.
- *On the performance of repaired classes*: looking at repair classes in the first part in Tbl. 3, we observe that the class of matching repairs is generally faster than the class of relevant repairs. i.e., median execution time drops from 49.7s to 28.4s from relevant to matching attempts, and its average time decreases from 408.6s to 77.9s from relevant to matching attempts. This indicates that outliers of repair attempts in the matching class either do not happen as frequently or do not take as much time as their relevant counterparts.

RQ2: What is the effect of the dubious location?

- *On the effect of fault inclusion on repaired classes*: we observe in the second part of Tbl. 3 that the repairs in the inclusive category are about twice as many as repairs of the non-inclusive category (from 1,325 to 677). Despite starting the experiment with an equal number of inclusive and non-inclusive attempts, more of the non-inclusive attempts fall into the unrepaired or no synthesis category, resulting in `ContractDR` generating fewer repairs for them. This indicates that if the dubious expression is in the right place, it is more likely to get to a matching repair. Furthermore, a small number of non-inclusive attempts can result in matching repairs, indicating that there are some locations outside of the occurrence of the fault that, if repaired, can compensate for the introduced fault.
- *On the effect of fault inclusion on performance*: in the middle part of Tbl. 3 we observe that the average execution of repairs generated from the non-inclusive repair attempts (68.9, and 64.3 in match and relevant repairs) are faster than repairs generated by the inclusive attempts (79.2, and 647.2 in match and relevant repairs). This indicates that there are fewer options in the not-inclusive class and therefore less search that `ContractDR` needs to do until it terminates; this led to the overall faster execution. Also, the average time of `SAT?` query in the inclusive, relevant repairs are much higher than other classes, indicating that relevant repairs can become computationally expensive to compute due to running multiple minimal repairs iterations.
- *On the effect of a poor sketch*: Although we have not extensively studied the impact of the sketch on the repair, its effect shows up in the no-synthesis attempts (no syn attempts), which indicates that `ContractDR` could not find a repair using the automatically composed repair generators. Tbl. 5 shows that these attempts tend to be relatively fast.

Table 5: Effect of poor sketch.

attempts#	4,060
no synth attempts#	698
median / avg exec. (s)	1.2 / 7.6
median / avg SAT (s)	0 / 1.7
median / avg VALID (s)	1.1 / 5.9

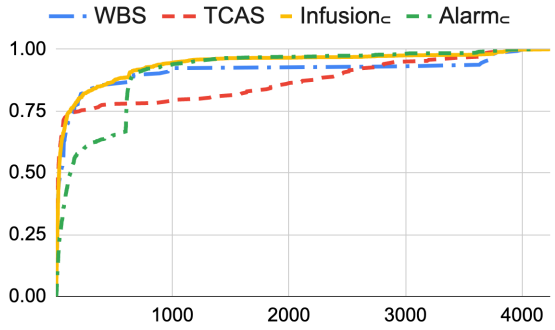

Figure 2: CDF Execution Time (s)

RQ3:What is the effect of a poor hypothesized contract?

The lower part of a Tbl. 3 shows the effect of poor hypothesized contract. There the matching repairs generated from 1-fault repair attempts are about a third higher than their 2-fault counterparts (303 to 206). Also, the total number of repairs (match, and relevant) between 1-fault and 2-fault attempts remain about the same (932 repairs in 1-fault, and 1,070 in 2-faults). This result is expected as `ContractDR`'s preference for a tighter repair does not include any guidance towards an ideal repair.

RQ4: What is the effect of the implementation's complexity?

In Tbl. 4, we observe that as the complexity and the size of the benchmark increases (from left to right), the likelihood of finding repairs decreases. `ContractDR`'s success rate of finding repairs ranged from 97% in WBS and TCAS, to 89% and 66% in Infusion$_c$ and Alarm$_c$. Notably, the relatively low repair percentage in the Alarm$_c$ benchmark is due to the non-linear computation in the benchmark. Also observe that reaching the minimal repair (Minimal Reached) is the most common termination reason in repaired attempts, indicating that `ContractDR` is finding the minimal repairs most of the time. The second most common termination reason with a repair is different among the benchmarks. For example, terminating due to timing out of the `SAT?` query (minimal `SAT?` TO) is the second termination reason for TCAS. This is because TCAS has the broadest range of integer values that `ContractDR` needs to search through to find a valid repair. However, terminating due to the validity query timing out (*minimal* `VALID?` *TO*) only shows up in Alarm$_c$ due to its non-linear computation. Terminating due to reaching the limit of invalid candidates (minimal cand. bound) is correlated with the size and the number of different variables in the mutated contract. As the number of variables and depth of the repair sketch increases, the number of candidates increases. Since Infusion$_c$ has the highest number of contracts (14 out of the 29 contracts) with many different variables, the likelihood of termination due to reaching the limit of invalid candidates increases. Finally, the overall *1-hour TO* is rarely reached.

The highest percentage of unrepaired mutants (34%) is for Alarm_c. This is again due to its non-linear computation. In general, although we have not implemented that, approximation of non-linear computation is likely to improve repair results for non-linear components. We leave that improvement for future work. Finally, we observe that the SAT? queries dominates most of the execution time. except for the Alarm_c where verifying contracts was the bottleneck. Fig. 2 shows the Cumulative Distribution Function (CDF) of the running time of ContractDR. Observe that the majority of the runs were fast, but a minority were much slower. The intersections of the curves with the 0.5 horizontal line correspond to median runtimes, thus indicating that the median runtime was under a minute or two for all the benchmarks. The runs between 0.5 and 1.00 also show that 60-90% of the runs were relatively fast. On the other hand, the top part of the graph shows that some runs took much longer. This is mostly associated with having a lot of iterations and finding many repairs. The nearly-vertical section of the Alarm curve a little past 600 seconds suggests that around 15-20% of all the Alarm runs took a similar amount of time, slightly more than 600 seconds. This is usually due to having small, inexpensive queries, followed by a complex VALID? query that timed out.

RQ5: Can dynamically-inferred invariants be used as repairs?

To compare repairs generated by ContractDR with invariant properties from previous techniques, we generated invariants for our 4 benchmarks and compared them with the original properties. We first tried to use iDiscovery [43] which is a feedback-driven invariant discovery tool that utilizes symbolic execution (SPF [33]) to search for new tests that might falsify Daikon invariants. However, its path exploration was too slow, which we attribute to using an old version of SPF. So instead we generated test cases from complete path exploration using the latest version of SPF [2], then used Daikon [12] to generate invariants from those tests. We used the maximum number of inputs for each benchmark for which SPF can finish complete path exploration in less than 24 hours (Tbl. 6). Due to the complexity of Alarm_c, we limited the exploration of the symbolic exploration tree to a depth of 152 for a single step. We also configured Daikon to generate set-of-values ("OneOf") invariants up to size 8, and we guided the invariant generation towards conditions that appear in the original properties. [8]

The results in Tbl. 6 shows that the majority of the invariants, similar to ContractDR's repairs except those on WBS, fall in the relevant repair class, but unlike ContractDR much fewer invariants (Inv) are generated in the matching class. Most of the original properties are implications where the antecedent is a conjunction, which is a grammatical form beyond what Daikon's algorithms generate. Daikon creates implications by searching for properties that hold on a subset ("split") of the data, but only when the antecedent always holds on one side of the split and never on the other; this requires that the antecedent be a single invariant [11, section 6.2]. Daikon could only generate such invariants if it found a single invariant equivalent to the conjunction. But we also attribute some of the unsatisfying results to limitations of the test cases we provided to Daikon; for instance Daikon generated a number of invariants (which we exclude from the evaluation) which held over our test cases but not in general.

[8] We manually created split info files that contains the conditions that Daikon should use to create conditional invariants.

Table 6: Classes of Daikon Invariants as Repairs

	inputs	time(m)	inv	match	relevant
WBS	15	204	14	0	14
TCAS	24	18	2	0	2
Infusion_c	100	141	46	2	44
Alarm_c (152 depth)	43	330	6	0	6
Targeted P9	100	-	28	1 (combined)	28

To further investigate, we conducted a targeted experiment. The result of this experiment shows in the last row in Tbl. 6. The goal of this experiment was to create a perfect test suite for a specific property, then compare the resulting invariants. We defined a perfect test suite for a property P, as the test suite that allows Daikon to generate invariants whose conjunction implies the targeted property. The property we were targeting was in the Infusion_c $(On \wedge Alarm = 4) \rightarrow (mode = 1 \vee mode = 6 \vee mode = 7)$. The result in the last row of Tbl. 6 shows that with a sufficient test suite, Daikon is successful in generating 28 invariants whose conjunction implies the human-written property. However, none of the individual invariants matches the human-written one, and conjoining 28 different invariants results in a combined property (comb) that is impractically complex compared to the desired property. When trying to find a minimum set of invariants where the implication would still hold, we found that the resultant invariants use 12 boolean terms, as opposed to only 5 boolean terms in the original property. This illustrates how a repair approach is more effective to create a compact and human-readable.

To conclude, Daikon is geared towards finding certain forms of invariants that hold over programs' execution, while ContractDR is intended to finding a single contract that is "close" to the desired. The experiment illustrates that even with substantial guidance, invariant generation with Daikon cannot be easily repurposed to address the problem that ContractDR addresses.

7 CONCLUSION AND FUTURE WORK

We define the problem of repairing contracts that are not valid on their implementations. We describe a general-purpose contract repair algorithm and show how to instantiate for it to repair contracts of reactive components. All algorithms are sound, boundedly minimal, and under reasonable assumptions terminating and complete. We presented an extensive study on the various factors affecting the performance and the quality of the generated repairs.

In the future, we plan to generate smarter sketches, by utilizing dependencies among the component's variables. We also plan on extending this work to prioritize repair locations. And finally, a follow-up user study would complement the results presented in this paper.

ACKNOWLEDGMENTS

The authors would like to thank An Nguyen for supplying benchmarks for evaluating the initial prototype of this work.

Also, the research described in this paper has been supported in part by the National Science Foundation under grant 1563920.

REFERENCES

[1] [n. d.]. *ContractDR GitHub Repository*. Retrieved March 29, 2022 from https://github.com/sohah/ContractDR

[2] [n. d.]. *jpf-symbc*. Retrieved March 29, 2022 from https://github.com/SymbolicPathFinder/jpf-symbc

[3] [n. d.]. *Lustre*. Retrieved March 29, 2022 from http://www-verimag.imag.fr/The-Lustre-Programming-Language-and

[4] [n. d.]. *Z3*. Retrieved March 29, 2022 from https://rise4fun.com/z3/tutorial

[5] Chris Ackermann, Rance Cleaveland, Samuel Huang, Arnab Ray, Charles Shelton, and Elizabeth Latronico. 2010. Automatic Requirement Extraction from Test Cases. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6418 LNCS (2010), 1–15. https://doi.org/10.1007/978-3-642-16612-9_1

[6] Rajeev Alur, Salar Moarref, and Ufuk Topcu. 2013. Counter-Strategy Guided Refinement of GR(1) Temporal Logic Specifications. *2013 Formal Methods in Computer-Aided Design, FMCAD 2013* (aug 2013), 26–33. arXiv:1308.4113 https://arxiv.org/abs/1308.4113v1

[7] David Arney, Raoul Jetley, Paul Jones, Insup Lee, and Oleg Sokolsky. 2007. Formal methods based development of a PCA infusion pump reference model: Generic infusion pump (GIP) project. *Proceedings - 2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability, HCMDSS/MDPnP 2007* (2007), 23–33. https://doi.org/10.1109/HCMDSS-MDPNP.2007.36

[8] P. E. Black, V. Okun, and Y. Yesha. 2000. Mutation operators for specifications. *Proceedings ASE 2000: 15th IEEE International Conference on Automated Software Engineering* (2000), 81–88. https://doi.org/10.1109/ASE.2000.873653

[9] Davide G. Cavezza. 2016. Interpolation-Based GR(1) Assumptions Refinement. *CoRR* abs/1611.0 (2016). http://arxiv.org/abs/1611.07803

[10] Hoang Duong, Thien Nguyen, and Satish Chandra. 2013. SemFix : Program Repair via Semantic Analysis. In *ICSE*. 772–781. https://compsec.comp.nus.edu.sg/papers/ICSE13-SEMFIX.pdf

[11] Michael D Ernst and Contributors. 2020. *{Daikon} User Manual* (version 5.8.4 ed.). https://plse.cs.washington.edu/daikon/download/doc/daikon.pdf

[12] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming* 69, 1-3 (dec 2007), 35–45. https://doi.org/10.1016/J.SCICO.2007.01.015

[13] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. 2018. The JKind Model Checker. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10982 LNCS (jul 2018), 20–27. https://doi.org/10.1007/978-3-319-96142-2_3

[14] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning invariants using decision trees and implication counterexamples. *ACM SIGPLAN Notices* 51, 1 (apr 2016), 499–512. https://doi.org/10.1145/2837614.2837664

[15] Patrice Godefroid and Daniel Luchaup. 2011. Automatic partial loop summarization in dynamic test generation. *2011 International Symposium on Software Testing and Analysis, ISSTA 2011 - Proceedings* (2011), 23–33. https://doi.org/10.1145/2001420.2001424

[16] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. SketchFix: A tool for automated program repair approach using lazy candidate generation. *ESEC/FSE 2018 - Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (oct 2018), 888–891. https://doi.org/10.1145/3236024.3264600

[17] Soha Hussein, Vaibhav Sharma, Stephen McCamant, Sanjai Rayadurgam, and Mats Heimdahl. 2021. Counterexample Guided Inductive Repair of Reactive Contracts. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1190–1192.

[18] Soha Hussein, Vaibhav Sharma, Michael W Whalen, Stephen McCamant, and Willem Visser. [n. d.]. JavaRanger. https://github.com/vaibhavbsharma/java-ranger

[19] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program Repair as a Game. *Lecture Notes in Computer Science* 3576 (2005), 226–238. https://doi.org/10.1007/11513988_23

[20] Aviv Kuvent, Shahar Maoz, and Jan Oliver Ringert. 2017. A symbolic justice violations transition system for unrealizable GR(1) specifications. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (aug 2017), 362–372. https://doi.org/10.1145/3106237.3106240

[21] Xuan Bach D. Le, Duc Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and semantic-guided repair synthesis via programming by examples. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (aug 2017), 593–604. https://doi.org/10.1145/3106237.3106309

[22] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. 2011. Mining assumptions for synthesis. *9th ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011* (2011), 43–50. https://doi.org/10.1109/MEMCOD.2011.5970509

[23] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. *2015 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2015 - Proceedings* (aug 2015), 166–178. https://doi.org/10.1145/2786805.2786811

[24] Shahar Maoz, Jan Oliver Ringert, and Rafi Shalom. 2019. Symbolic Repairs for GR(1) Specifications. *Proceedings - International Conference on Software Engineering* 2019-May (may 2019), 1016–1026. https://doi.org/10.1109/ICSE.2019.00106

[25] Matias Martinez and Martin Monperrus. 2016. ASTOR: A program repair library for Java (Demo). *ISSTA 2016 - Proceedings of the 25th International Symposium on Software Testing and Analysis* (jul 2016), 441–444. https://doi.org/10.1145/2931037.2948705

[26] Sergey Mechtaev, Manh Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. *Proceedings - International Conference on Software Engineering* 2018-May (aug 2018), 129–139. https://doi.org/10.1145/3180155.3180247

[27] Anitha Murugesan, Michael W. Whalen, Sanjai Rayadurgam, and Mats P.E. Heimdahl. 2013. Compositional verification of a medical device system. *HILT 2013 - Proceedings of the ACM Conference on High Integrity Language Technology* (2013), 51–64. https://doi.org/10.1145/2527269.2527272

[28] ThanhVu Nguyen, Matthew B Dwyer, and Willem Visser. 2017. SymInfer: Inferring Program Invariants Using Symbolic States. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, 804–814.

[29] Thanhvu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. 2014. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Transactions on Software Engineering and Methodology* 23, 4 (sep 2014). https://doi.org/10.1145/2556782

[30] Thanh Vu Nguyen, Timos Antonopoulos, Andrew Ruef, and Michael Hicks. 2017. Counterexample-guided approach to finding numerical invariants. *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (aug 2017), 605–615. https://doi.org/10.1145/3106237.3106281

[31] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. ACM, New York, NY, USA, 257–271. https://doi.org/10.1145/93542.93578

[32] Saswat Padhi, Rahul Sharma, and Todd Millstein. 2016. Data-Driven precondition inference with learned features. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 13-17-June (jun 2016), 42–56. https://doi.org/10.1145/2908080.2908099

[33] Corina S. Păsăreanu and Neha Rungta. 2010. Symbolic PathFinder: Symbolic Execution of Java Bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering* (Antwerp, Belgium) *(ASE '10)*. Association for Computing Machinery, New York, NY, USA, 179–180.

[34] Christoph Schulze and Rance Cleaveland. 2017. Improving Invariant Mining via Static Analysis. *ACM Trans. Embed. Comput. Syst* 16, 167 (2017). https://doi.org/10.1145/3126504

[35] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS'18)*. Curran Associates Inc., Red Hook, NY, USA, 7762–7773.

[36] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial sketching for finite programs. *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS* (2006), 404–415. https://doi.org/10.1145/1168857.1168907

[37] Saurabh Srivastava and Sumit Gulwani. 2009. Program verification using templates over predicate abstraction. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2009), 223–234. https://doi.org/10.1145/1542476.1542501

[38] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2013. Template-based program verification and program synthesis. *International Journal on Software Tools for Technology Transfer* 15, 5-6 (oct 2013), 497–518. https://doi.org/10.1007/S10009-012-0223-4

[39] David Trabish, Shachar Itzhaky, and Noam Rinetzky. 2021. A bounded symbolic-size model for symbolic execution. *ESEC/FSE 2021 - Proceedings of the 29th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (aug 2021), 1190–1201. https://doi.org/10.1145/3468264.3468596

[40] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 47–55. https://doi.org/10.1145/207110.207115

[41] Michael Whalen, Darren Cofer, Steven Miller, Bruce H Krogh, and Walter Storm. 2007. Integration of Formal Analysis into a Model-Based Software Development Process. In *Proceedings of the 12th International Conference on Formal Methods for Industrial Critical Systems (FMICS'07)*. Springer-Verlag, Berlin, Heidelberg,

68–84.

[42] Michael W. Whalen, Andrew Gacek, Darren Cofer, Anitha Murugesan, Mats P.E. Heimdahl, and Sanjai Rayadurgam. 2013. Your 'what' is my 'how': Iteration and hierarchy in system design. *IEEE Software* 30, 2 (2013), 54–60. https://doi.org/10.1109/MS.2012.173

[43] Lingming Zhang, Guowei Yang, Neha Rungta, Suzette Person, and Sarfraz Khurshid. 2014. Feedback-driven dynamic invariant discovery. *2014 International Symposium on Software Testing and Analysis, ISSTA 2014 - Proceedings* (jul 2014), 362–372. https://doi.org/10.1145/2610384.2610389