Loop Analysis and Repair

Nafi Diallo, PhD Candidate
Department of Computer Science
Advisor: Ali Mili

New Jersey Institute of Technology

Abstract. This doctoral work proposes to use invariant relations to analyze and repair loops. We discuss how invariant relations allow us to derive loop properties such as termination, correctness and incorrectness and to generate invariant assertions. We also present a method to statically repair a loop using invariant relations, to what we refer as "Debugging without Testing".

1 Introduction

Software quality is of critical importance due to the pervasiveness of software in modern societies, its use in critical applications and its growing size and complexity. The demands on software technology are putting relentless pressure on software research to deliver ever more capable methods, tools, and processes.
For imperative programming languages,still most prevalent in software production, loops still constitute a main source of complexity and to analyze a loop, one must re-engineer the inductive argument that underlies its construction.
Due to the usual difficulty of this task, many approaches proceed by unrolling the loop a number of times, and analyzing the resulting program as a sequential code. In this work, we explore an orthogonal approach based on the concept of invariant relations, which allow approximating the function of a loop. The choice between capturing all the functional details of a loop whose iterations are bounded, and approximating its function for all possible executions, can be viewed as a trade-off between knowing everything about some executions and knowing something about all executions. We argue for the latter approach on the grounds that knowing everything is not necessary (many properties of interest can be established with partial information) and that making claims about bounded executions is not sufficient (a property may hold for bounded executions and fail to hold for unbounded executions).
This work contributes the following:

- A definition of the concept of loop convergence as the integration of termination and abort-freedom
- Proofs of correctness and incorrectness of while loops
- A method to statically prove that a fault is removed and that the resulting program is more correct.

– An automated loop analysis tool via analysis of the source code (to compute artifacts such as convergence condition, loop function, correctness verification)

In the following sections, we discuss related work, describe the proposed methods, and present a description of the plan of evaluation of this work.

2 Background and Related Work

This work is part of an ongoing project. [13] propose to use the relational approach for automated loop function computation via static analysis of the source code. In this context, they provide a definition of invariant relations and how it can be applied to compute the loop function and pre/post conditions.
This work builds on their results and investigates the use of invariant relations for loop analysis and repair with the aim to develop an automated tool for invariant relation generation and the implementation of the proposed methods.

2.1 Loop Analysis

Analysis of loop termination is a very mature area of research starting with the work of Alan Turing. This area can be characterized by a separation between two concerns: termination as a finite number of iterations and termination as abort-freedom. Termination as a finite number of iterations, which has also received the most attention, involves the discovery of well-founded ranking functions, namely transition invariants [2, 15] and size-change graphs [9]. In [2], Cook et. al. give a comprehensive survey of loop termination, in which they discuss transition invariants which are approximations of $(T \cap B)^+$ while invariant relations are approximations of $(T \cap B)^*$. This slight difference of form has a significant impact on the properties and uses of these distinct concepts. While transition invariants are used by Cook et al. to characterize the well founded property of $(T \cap B)^+$, we use invariant relations to approximate the function of a loop, and its domain. Abstract interpretation [3], Model Checking and Bounded Model Checking [6] are techniques aimed at capturing aspects of abort-freedom. Abstract interpretation is a broad scoped technique used to infer properties of programs by successive approximations of their execution traces and thus resembles most our approach. Model checking consists of exhaustively traversing all the reachable states of the system to verify the desired properties. Bounded model checking is a specialization of model checking in which the traversal is halted after a given number of iterations, in which case it is decided that no counter example exists and the property holds.
Overall, our work distinguishes itself with the approaches described above in that, with invariant relations, we can model the termination of while loops in a broad sense; we do that by merging the condition that the number of iterations is finite and the condition that every single iteration executes without causing an abort.

Traditionally, proof of correctness of loops involves two aspects: proof of partial correctness in the sense of Floyd/Hoare Logic and termination [7]. Another approach is based the weakest precondition theory [5].

2.2 Loop Repair

Most existing repair approaches rely on execution traces to identify faults. [8, 10, 11].
In [12] the authors consider an original program P and a variation P' of P, extract semantic information from P, and use it to instrument P' (by means of executable assertions). They then reason about semantic guarantees which can be inferred about the instrumented version of P' and analyze the condition under which both programs can execute without causing an abort (due to attempting an illegal operation), which they approximate by sufficient conditions and necessary conditions. They implement the VMV (*Verification Modulo Versions*) system aimed at exploiting semantic information about P in the analysis of P', and ensuring that the transition from P to P' happens without regression to decide that P' is *correct relative to* P. Their definition of relative correctness differs from the approach of this work, in several aspects: whereas [12] talk about relative correctness between an original program and a subsequent version in the context of adaptive maintenance (where P and P' may be subject to distinct requirements), we talk about relative correctness between an original (faulty) software product and a revised version of the program (possibly still faulty yet more-correct) in the context of corrective maintenance with respect to a fixed requirements specification; whereas [12] use a set of assertions inserted throughout the program as a specification, we use a relation that maps initial states to final states to specify the standards against which absolute correctness and relative correctness is defined; whereas [12] represents program executions by execution traces, we represent program executions by functions mapping initial states into final states; finally, while [12] define a successful execution as a trace that satisfies all the relevant assertions, we define a successful one as simply an initial state/ final state pair that falls with the specification (relation).
In [8] Lahiri et al. introduce *Differential Assertion Checking* for verifying the relative correctness of a program with respect to a previous version of the program. They explore applications of this technique as a tradeoff between soundness (which they concede) and lower costs (which they hope to achieve). Like the approach of the authors of [12] (from the same team), their work uses executable assertions as specifications, represents executions by execution traces, defines successful executions as traces that satisfy all the executable assertions, and targets abort-freedom as the main focus of the executable assertions. They define relative correctness between programs P and P' as the property that P' has a larger set of successful traces and a smallest set of unsuccessful traces than P; and they introduce relative specifications as specifications that capture functionality of P' that P does not have. Our approach differs from [8] in that we reason in terms of the initial and final states, characterize correct executions

by such pairs that belong to the specification, and we make no distinction between abort-freedom and normal functional properties.
In [11], the authors introduce a definition of relative correctness similar to that of [8]. Programs are modeled with trace semantics, and execution traces are compared in terms of executable assertions inserted into P and P'; in order for the comparison to make sense, programs P and P' have to have the same (or similar) structure and/or there must be a mapping from traces of P to traces of P'. When P' is obtained from P by a transformation, and when P' is provably correct relative to P, the transformation in question is called a *verified repair*. The authors introduce an algorithm specialized in deriving program repairs from a predefined catalog targeted to specific program constructs, such as: contracts, initializations, guards, floating point comparisons, etc. Similarly to ([12, 8]), the authors model programs by execution traces and distinguish between two types of failures: contract violations, when functional properties are not satisfied; and run-time errors, when the execution causes an abort; for the reasons we discuss above, we do not make this distinction, and model the two aspects with the same relational framework. They implement their approach in an automated tool based on the static analyzer cccheck, and assess their tool for effectiveness and efficiency.
In [14], Nguyen et al. present an automated repair method based on symbolic execution, constraint solving, and program synthesis; they call their method SemFix, on the grounds that it performs program repair by means of semantic analysis. This method combines three techniques: fault isolation by means of statistical analysis of the possible suspect statements; statement-level specification inference, whereby a local specification is inferred from the global specification and the product structure; and program synthesis, whereby a corrected statement is computed from the local specification inferred in the previous step. The method is organized in such a way that program synthesis is modeled as a search problem under constraints, and possible correct statements are inspected in the order of increasing complexity. When programs are repaired by SemFix, they are tested for (absolute) correctness against some predefined test data suite; as we argue throughout [4], it is not sensible to test a program for absolute correctness after a repair, unless we have reason to believe that the fault we have just repaired is the last fault of the program (how do we ever know that?). By advocating to test for relative correctness, we enable the tester to focus on one fault at a time, and ensure that other faults do not interfere with our assessment of whether the fault under consideration has or has not been repaired adequately.

3 Semantics

We assume the reader familiar with elementary relational mathematics and generally use the notation adapted from [1].
Given a set S defined by the values of some program variables, say x and y, elements of S are denoted by s and expressed as $s = \langle x, y \rangle$. We represent the

x-component and (resp.) y-component of s as $x(s)$ and $y(s)$. When there is no ambiguity, we refer to $x(s)$ as x and $x(s')$ as x' for elements s and s' of S. We refer to S as the state *space* of the program and to elements of S, denoted by s, as the *states* of the program.
A relation on S is a subset of the Cartesian product $S \times S$.
Of interest to us are the following constant relations on some set S:

- the *universal* relation, denoted by $L = S \times S$,
- the *identity* relation, denoted by $I = \{(s, s')|s = s'\}$,
- and the *empty* relation, denoted by ϕ.

Given that by definition relations are sets, set theoretic operations such as union($\cup$), intersection($\cap$) and complement($\overline{R}$, for a relation R) apply to them.
Operations on relations also include:

- the domain, *dom*(R), of R defined by *dom*$(R) = \{s|\exists s' : (s, s') \in R\}$,
- the *range*, *rng*(R), of R, defined *rng*$(R) = \{s'|\exists s : (s, s') \in R\}$,.
- The *converse*, denoted by $\widehat{R}$, and defined by $\widehat{R} = \{(s, s')|(s', s) \in R\}$.
- The *product* of relations, say R and R', denoted by $R \circ R'$ (or RR') and defined by $R \circ R' = \{(s, s')|\exists s'' : (s, s'') \in R \wedge (s'', s') \in R'\}$.
- The *nucleus* of relation R, denoted by $\mu(R)$ and defined by $\mu(R) = R\widehat{R}$.
- The n^{th} *power* of relation R, for natural number n, denoted by R^n and defined by $R^0 = I$, and $R^n = R \circ R^{n-1}$, for $n \geq 1$.
- The *transitive closure* of relation R, denoted by R^+ and defined by $R^+ = \{(s, s')|\exists i > 0 : (s, s') \in R^i\}$.
- The *reflexive transitive closure* of relation R, denoted by R^* and defined by $R^* = I \cup R^+$. We admit without proof that $R^*R^* = R^*$ and that $R^*R^+ = R^+R^* = R^+$.
- The *pre-restriction* (resp. *post-restriction*) of relation R to predicate t is the relation $\{(s, s')|t(s) \wedge (s, s') \in R\}$ (resp. $\{(s, s')|(s, s') \in R \wedge t(s')\}$).

Given a program p on state space S, we let P be the function of p. P is defined as:
$P = \{(s, s')|p\ starts\ execution\ on\ s,\ then\ it\ terminates\ normally\ in\ state\ s'\}$
Normal termination means that the program terminates after a finite number of operations, without causing an abort and returns a well-defined final state.
We consider while loops written in some C-like programming language, of the form `while (t) {b}` and the semantic following definition:
$[\{$`while (t) {b}`$\}] \equiv (T \cap B)^* \cap \widehat{\overline{T}}$.
B is the function of b and T is the vector defined by: $\{(s, s')|t(s)\}$

4 Invariant Relations

Informally, an invariant relation can be described as a binary relation between input states and their corresponding output states obtained by applying zero or more iterations of the loop body. More formally, we define an invariant relation as follows:

Definition 1. *Given a while loop of the form* w =`while t {b}` *on space* S*, a relation* R *on* S *is said to be an* invariant relation *for* w *if and only if it is a reflexive and transitive superset of* $(T \cap B)$.

To illustrate the concept of invariant relation, we consider the loop below on state space $S = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ where $\mathbb{N}$ is the set of natural numbers. A state $s \in S$ is defined by the integer variables n, f, and k such that a state $s =< n, f, k >$.

```
while (k!=n) {k=k+1; f=f*k;}
```

We consider the following relation: $R = \left\{(s, s') | \frac{f}{k!} = \frac{f'}{k'!}\right\}$.
This relation is reflexive and transitive, since it is the nucleus of a function; to prove that it is a superset of $(T \cap B)$ we compute the intersection $R \cap (T \cap B)$ and easily find that it equals $(T \cap B)$.
Other invariant relations include:
$R_1 = \{(s, s') | n' = n\}$ $R_2 = \{(s, s') | k \leq k'\}$.
One of the main attributes of invariant relations is that they allow us to derive invariant assertions, a key concept in the established approach for proving correctness of while loops. In [7], an invariant assertion α for a while loop w: `while t {b}` with respect to precondition ϕ and postcondition ψ is defined as a predicate on S such that the following are true:

- $\phi \Rightarrow \alpha$,
- $\{\alpha \wedge t\} b \{\alpha\}$,
- $\alpha \wedge \neg t \Rightarrow \psi$.

We consider the factorial example given above to which we add initialization of the variables involved.

```
w: n=n0;k=0;f=1; while (k!=n) {k=k+1; f=f*k;}
```

The reader can easily verify that $f = k!$ and $n = n0$ are invariant assertions for the loop with respect to the precondition $n = n0; f = 1; k = 1;$ and postcondition $f = n0!$
As expressed above, an invariant assertion depends on both the loop body and the context of the loop, namely its precondition and its post condition while invariant relations only involve the loop body.To make the comparison between invariant assertions and invariant relations meaningful, we redefine invariant assertions to be assertions that satisfy the condition $\{\alpha \wedge t\} b \{\alpha\}$ since it is the only condition that depends exclusively on the loop and does not depend on the precondition (as $\psi \Rightarrow \alpha$) nor the postcondition (as $\alpha \wedge \neg t \Rightarrow \psi$). Given a predicate α, let A be defined as a vector on S by: $A = \{(s, s') | \alpha(s)\}$

Definition 2. *Vector* A *is said to be an* invariant assertion *for the while loop* w*:* `while t {b}` *if and only if it satisfies the following condition:* $(A \cap T \cap B) \subseteq \widehat{A}$ *where* T *is a vector defined by predicate* t

The following two propositions, describe the relationships between invariants assertions and invariant relations.

Proposition 1. *Let R be an invariant relation of w:* `while t {b}` *on space S and let C be an arbitrary vector on S. Then $\widehat{R}C$ is an invariant assertion for w.*

Proposition 2. *Given an invariant assertion A, there exists an invariant relation R and a vector C such that $A = \widehat{R}C$.*

5 Loop Analysis

5.1 Convergence

In the spirit of merging the two aspects of termination described in related work, we introduce the concept of convergence as the integration of these two aspects. The following theorem provides a general framework for convergence and illustrates how we only need to model one aspect or another or both by our choice of invariant relation.

Theorem 1. *We consider a while loop w of the form* `w: while (t) {b}` *on space S, and we let R be an invariant relation for w. Then:* $WL \subseteq R\overline{T}$.

While this result provides that any invariant relation gives a necessary condition of termination, smaller invariant relations are favored as they can lead to both a necessary and sufficient condition. In [4], we present a theorem that provides means of capturing aspects of abort-freedom by generating invariant relations of this form : $R = \{(s, s')|\forall u : (s, u) \in B'^{*} \wedge (u, s') \in B'^{+} \Rightarrow u \in dom(B)\}$, where B' is a superset of B. In practice, B' is an approximation of B, derived by focusing on the variables that are involved in abort-prone statements and recording how B transforms them while $dom(B)$ is modeled using the abort condition of interest. For example, to model

- the condition that arithmetic operations in B does not cause overflow, $dom(B)$ will express a clause to the effect that all operations produce a result within the range of representable values.
- the condition of non-zero division in the execution of B, a condition is added in $dom(B)$ that ensures that all divisors in B are non-zero;

To illustrate the concept of convergence, we consider the following example where we assume that we want to avoid a division by zero (variable j).

```
while (i!=0) {i=i-1; j=j+1; k=k-k/j;}
```

We find the following invariant relations:

- $R_1 = \{(s, s')|j \leq j'\}$
- $R_2 = \{(s, s')|i \geq i'\}$
- $R_3 = \{(s, s')|i + j == iP + jP\}$
- $R_4 = \{(s, s')|\forall h : j \leq h < j', 1 + h \neq 0\}$

Using these relations, we compute the following convergence condition:
$(i = 0 \vee (i \geq 1 \wedge j \geq 0) \vee (i > 0 \wedge 1 + i + j \leq 0))$

5.2 Correctness Verification

In [4], two propositions are introduced for computing necessary and sufficient conditions of correctness using invariant relations. We use them to derive an algorithm which generates successive invariant relations, and tests a necessary condition of correctness and a sufficient condition of correctness, until one of three conditions arises:

- either the sufficient condition is true, then we diagnose the loop as *correct*.
- or the necessary condition is false, then we diagnose the loop as *incorrect*.
- or we run out of invariant relations before we reach the conclusions above; in which case we exit with the conclusion that we do not know enough about the loop to rule on its correctness.

6 Loop Repair

Simply put, loop repair consists of removing a fault and proving that the fault has been removed. Our proposed method for loop repair consists of the following.

1. Observation of failure (loop is diagnosed as incorrect by finding an invariant relation that violates the necessary condition of correctness)
2. fault diagnosis (statements involving variables of the invariant relation above are targeted)
3. fault removal (deriving a new invariant that verifies the necessary condition of correctness)
4. proof of relative correctness (defined below)

6.1 Relative Correctness

Definition 3. *Let R be a specification on state space S and let p and p' be two programs on space S whose functions are respectively P and P'.*

- *We say that program p' is* more-correct *than program p with respect to specification R (abbreviated by: $P' \sqsupseteq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$.*
- *Also, we say that program p' is* strictly more-correct *than program p with respect to specification R (abbreviated by: $P' \sqsupset_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.*

where L be the universal relation on S

$(R \cap P)L$ is referred to as the *competence domain* of program p; it represents the set of initial states for which the program agrees with the specification R. Thus to be more-correct means to have a larger competence domain. Note for program p' to be more-correct than program p, it does not have to duplicate the behavior of p over the competence domain of p: it may have a different behavior (since R is potentially non-deterministic) provided this behavior is also correct with respect to R; In [4], We illustrate the concept with an example for which $(R \cap P)L = \{1, 2, 3, 4\} \times S$, $(R \cap P')L = \{1, 2, 3, 4, 5\} \times S$, where $S = \{0, 1, 2, 3, 4, 5, 6\}$. Hence p' is more-correct than p with respect to R.

7 Proving Relative Correctness for Loops

In [4], we propose a theorem and an algorithm to the effect that we can achieve the four steps mentioned above for loop repair. To illustrate loop repair, we consider the following loop, where all the variables except `t` are of type `double`, and where a and b are positive constants.

```
w:  while (abs(r-p)>upsilon)
      {t=t+1;n=n+x; m=m-1;l=l*(1+b);k=k+1000;y=n+k;
       w=w+z;z=(1+a)+z;v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

We consider the following specification:

$$R = \{(s,s') | b < a < 1 \wedge x' = x \wedge w' = w - z \times \frac{1-(1+a)^{t'-t}}{a}$$
$$\wedge k' = k + 1000 \times (t'-t) \wedge t \leq t' \wedge 0 < l \leq l' \wedge z > 0 \wedge l \times (1+b)^{-t} = l' \times (1+b)^{-t'}\}.$$

Analysis of this loop by our invariant relation generator derives fourteen invariant relations, five of which are found to be incompatible with the specification. We select the following incompatible invariant relation for remediation: $Q = \{(s,s') | l \times (1+b)^{-\frac{z}{1+a}} = l' \times (1+b)^{-\frac{z'}{1+a}}\}$. To fix this incompatibility, we must alter variable z or variable l. We compute the condition on z and l under which a change in these variables does not alter any of the relevant compatible relations, and we find: $z' \geq z \wedge (l = l' \vee l \times (l' - l) > 0)$. z is chosen and common mutation operators are applied to the statement $\{$`z=(1+a)+z`$\}$ while preserving the condition $z' \geq z$; for each mutant of this statement, we recompute the new invariant relation that substitutes for Q and check whether it is compatible with R. We find that the statement $\{$`z=(1+a)*z`$\}$ produces a compatible invariant relation, and conclude that the loop w' obtained when we replace $\{$`z=(1+a)+z`$\}$ by $\{$`z=(1+a)*z`$\}$ is more-correct than w with respect to R. Running the invariant relations generator on the new loop produces fourteen invariant relations, one of which is incompatible with R (hence w' is indeed incorrect); it seems that by removing the earlier fault we have remedied four invariant relations at once. Applying the same process to w', we find the following loop, which is correct with respect to R:

```
wc: while (abs(r-p)>upsilon)
{t=t+1;n=n+x; m=m+1;l=l*(1+b);
k=k+1000;y=n+k;w=w+z;z=(1+a)*z;
v=w+k;r=(v-y)/y;u=(m-n)/n;d=r-u;}
```

8 Conclusions and Prospects

Presently, we are developing and evolving the tool to automate the methods described above. It currently covers numeric data types and provide artifacts related to convergence and correctness verification. The results of the experiments are promising and encouraging. We intend to assess the effectiveness of

the proposed approach and tool by comparing the tool with tools from other approaches[Astree, Genprog],using relevant benchmarks. While evolving the tool to include more application domains, we plan to further explore the implications and applications of relative correctness, to further derive techniques for proving relative correctness by static analysis of the source code. We also intend to create a new benchmark for convergence and analysis of program repair with relative correctness. This should all contribute as evidence of the significance of this work.

References

1. C. Brink, W. Kahl, and G. Schmidt. *Relational Methods in Computer Science*. Springer Verlag, New York, NY and Heidelberg, Germany, 1997.
2. B. Cook, A. Podelski, and A. Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5), 2011.
3. P. Cousot. Abstract interpretation. Technical Report www.di.ens.fr/čousot/AI/, Ecole Normale Superieure, Paris, France, August 2008.
4. N. Diallo, W. Ghardallou, and A. Mili. Correctness and relative correctness. In *Proceedings, 37th International Conference on Software Engineering*, Firenze, Italy, May 20–22 2015.
5. E. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
6. S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *VSTTE*, pages 261–277, 2012.
7. C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576 – 583, Oct. 1969.
8. S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel. Differential assertion checking. In *Proceedings, ESEC/ SIGSOFT FSE*, pages 345–455, 2013.
9. C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *ACM SIGPLAN Notices*, volume 36, pages 81–92. ACM, 2001.
10. C. LeGoues, S. Forrest, and W. Weimer. Current challenges in automatic software repair. *Software Quality Journal*, 21(3):421–443, 2013.
11. F. Logozzo and T. Ball. Modular and verified automatic program repair. In *Proceedings, OOPSLA*, pages 133–146, 2012.
12. F. Logozzo, S. Lahiri, M. Faehndrich, and S. Blackshear. Verification modulo versions: Towards usable verification. In *Proceedings, PLDI*, 2014.
13. A. Mili, S. Aharon, and C. Nadkarni. Mathematics for reasoning about loop. *Science of Computer Programming*, pages 989–1020, 2009.
14. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings, ICSE*, pages 772–781, 2013.
15. A. Podelski and A. Rybalchenko. Transition invariants. In *Proceedings, 19th Annual Symposium on Logic in Computer Science*, pages 132–144, 2004.