A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows*

Dominic Steinhöfel and Nathan Wasser

TU Darmstadt, Dept. of Computer Science, Darmstadt, Germany
steinhoefel@cs.tu-darmstadt.de, nate@sharpmind.de

Abstract. Invariants are a standard concept for reasoning about unbounded loops since Floyd-Hoare logic in the late 1960s. For real-world languages like Java, loop invariant rules tend to become extremely complex. The main reason is non-standard control flow induced by return, throw, break, and continue statements, possibly combined and nested inside inner loops and try blocks. We propose the concept of a *loop scope* which gives rise to a new approach for the design of invariant rules. This permits "sandboxed" deduction-based symbolic execution of loop bodies which in turn allows a modular analysis even of complex loops. Based on the new concept we designed a loop invariant rule for Java that has full language coverage and implemented it in the program verification system KeY. Its main advantages are (1) much increased comprehensibility, which made it considerably easier to argue for its soundness, (2) simpler and easier to understand proof obligations, (3) a substantially decreased number of symbolic execution steps and sizes of resulting proofs in a representative set of experiments. We also show that the new rule, in combination with fully automatic symbolic state merging, realizes even greater proof size reduction and helps to address the state explosion problem of symbolic execution.

1 Introduction

In the past decades, *deductive software verification* [9] techniques evolved from theoretical approaches reasoning about simple while languages [13] to systems such as Spec# [2], Frama-C [7], OpenJML [6] and KeY [1] which are capable of proving complex properties about programs in industrial programming languages such as C, C# and Java [16,11]. Naturally, the complexity of the languages is reflected in the complexity of the verification, raising the question: How can we adequately handle language complexity, while restraining the negative impact of overly complex verification procedures on comprehensibility and performance?

Prominent deductive verification techniques comprise verification condition generation and Symbolic Execution (SE). The scope of this work is the latter.

* **The final authenticated version is available online at** https://doi.org/10.1007/978-3-319-66845-1_18.

As opposed to concrete execution, SE [8] treats inputs to a program as abstract symbols as long as they are not assigned a concrete value; thus, programs can be analyzed for all possible input values. Whenever the execution depends on the concrete value of a symbolic variable, it makes a case distinction, following each possible branch independently. The outcome of SE is a Symbolic Execution Tree (SET). We distinguish two types of SE approaches: (1) *Lightweight* SE has its applications in bug finding or, for instance, concolic testing [14]. Programs are instrumented by replacing data types with symbolic representations or by the addition of function calls to the SE engine, which is in turn backed by an external SMT solver. Lightweight SE has been employed in the analysis of whole software libraries [4]. Example systems include KLEE [4] and Java PathFinder [17]. (2) *Heavyweight* SE can be used to prove complex functional properties about programs which are executed by a symbolic interpreter. A strong focus is put on *modularity*: e.g., single methods may be thoroughly analyzed independently from the concrete code of others. To achieve this, the analysis depends on specifications such as method contracts and loop invariants. Heavyweight SE systems can rely on an external solver, or be integrated with an internal theorem proving engine. Due to high computation time and the effort required for creating specifications, they do not scale to complete libraries, and are instead employed to assert strong guarantees about critical routines [11] or to build powerful tools like symbolic debuggers [12]. Example systems encompass KeY [1], VeriFast [21] and KIV [20]. In this paper, we consider heavyweight SE.

Heavyweight SE is strongly affected by both the performance and comprehensibility aspects phrased in the question at the beginning: The number of branches in an SET grows exponentially in the number of static branching points in the analyzed program, which is referred to as the *path explosion problem* in literature [5]. Additionally, proving the validity of complex properties may require interaction with the prover, for which it is essential that the proof is transparent and understandable to the user.

For reasoning about unbounded loops, invariants are standard since Hoare logic [13] and play a central role in heavyweight SE systems. This paper pushes forward a new kind of *loop invariant rule* tackling the aforementioned problems by integrating a novel program abstraction, which we refer to as *loop scopes*, and an automatic predicate abstraction-based state merging technique exploiting existing specification elements for infering predicates while maintaining precision.

Standard loop invariant rules require certain contorted maneuvers to deal with abnormal control flow induced, e.g., by breaks and exceptional behavior; these measures include non-trivial code transformation or a regime based on a multitude of artificial flags. Our approach avoids this by realizing a "sandboxing" technique: Loop bodies are encapsulated inside loop scopes, the semantics of which allow for a graceful and modular handling of nested loops and complex, irregular control flow. The loop bodies themselves do not have to be changed. Our implementation and evaluation for the heavyweight SE system KeY demonstrates that the loop scope invariant rule contributes to significantly shorter SETs that

are moreover better understandable for a human observer. The integration of state merging helps to reduce proof sizes even further.

The idea of loop scopes appeared first in [22] and is not yet published. Our additional contributions are (1) a definition of the semantics of loop scopes and an outline of a soundness proof for the invariant rule (Sect. 3), (2) the implementation and experimental evaluation of the rule (Sect. 4), and (3) a predicate abstraction-based approach for merging SE states arising from the execution of loops with non-standard control flow (Sect. 5).

2 Program Logic for Symbolic Execution

One convenient approach to concisely describe heavyweight SE is the formalization of SE steps as rules in a formal calculus. For expressing our concepts, we chose Java Dynamic Logic (JavaDL) [1], a program logic for Java (the main concepts of which can be straightforwardly extended to other sequential languages like C#). JavaDL is an extension of first-order logic for formulating assertions about program behavior; programs and formulas are integrated within the same language. To this end, JavaDL contains *modalities* for expressing partial and total correctness, where the latter also includes proving that the program terminates. For simplicity, we restrict ourselves to the former in this paper: $[p]\varphi$ expresses that *if* the program p terminates, then the formula φ holds.

The JavaDL calculus is a *sequent calculus* in which, as usual, rules consist of one conclusion and at least one premise, and are applied bottom-up. The SE rules of the calculus operate on the first *active* statement *stmt* in a modality $[\pi\ stmt\ \omega]$. The nonactive prefix π consists of sequences of opening braces, beginnings "`try {`" of `try-catch-finally` blocks, or special constructs like the loop scopes introduced in this paper. The postfix ω denotes the "rest" of the program; in particular, it contains closing braces corresponding to the opening braces in π.

Example 1. Consider the following modality, where the active statement `i=0;` is wrapped in a labeled `try-finally` block, and the nonactive prefix π and the "rest" ω are the indicated parts of the program:

$$[\underbrace{\texttt{l:\{try \{}}_{\pi}\ \underbrace{\texttt{i = 0;}}_{stmt}\ \underbrace{\texttt{j = 0;\} finally \{k = 0;\}\}}}_{\omega}]$$

The sequent $\texttt{i} < 0 \vdash [\pi\ \texttt{i=0;}\ \omega](\texttt{i} \doteq 0)$, embedding this modality, intuitively expresses "when started in a state where `i` is negative, 'π `i=0;` ω' either does not terminate, or terminates in a state where `i` is zero (since Java is deterministic)". The SE rule applicable to the sequent, assignment, transforms the active statement into a state-changing *update*. Below, we show the definition of this rule on the right and its application on the sequent on the left (Γ and Δ are placeholders for sets of formulas):

$$\frac{\texttt{i} < 0 \vdash \{\texttt{i} := 0\}[\pi\ \omega](\texttt{i} \doteq 0)}{\texttt{i} < 0 \vdash [\pi\ \texttt{i=0;}\ \omega](\texttt{i} \doteq 0)} \qquad \left[\begin{array}{c}\textsf{assignment}\\ \dfrac{\Gamma \vdash \{\texttt{x} := expr\}[\pi\ \omega]\varphi, \Delta}{\Gamma \vdash [\pi\ \texttt{x=}expr\texttt{;}\ \omega]\varphi, \Delta}\end{array}\right]$$

The above example employed another syntactical category of JavaDL called *updates*, which denote state changes. *Elementary* updates $\mathtt{x} := t$ syntactically represent the states where the program variable $\mathtt{x}$ attains the value of the term t. Updates can be combined to *parallel* updates $\mathtt{x} := t_1 \,||\, \mathtt{y} := t_2$, and can be *applied* to terms and formulas, where we write $\{\mathcal{U}\}\varphi$ for applying the update $\mathcal{U}$ to the formula φ. Semantically, φ is then evaluated in the state represented by $\mathcal{U}$. For a full account of JavaDL, we refer the reader to [1].

3 The Loop Invariant Rules

In the verification of sequential programs, and also in SE, the treatment of loops is one of the most crucial issues. Loops with a fixed upper bound on the number of iterations can be handled by *unwinding*. Whenever this bound is not known a priori, often *loop invariant rules* are employed. The "classic" loop invariant rule has the following shape [1,8], where *Inv* is a supplied loop invariant:

loopInvariant

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}\, Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}((Inv \wedge se \doteq TRUE) \rightarrow [body]Inv), \Delta & \text{(preserved)}\\ \Gamma \vdash \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}((Inv \wedge se \doteq FALSE) \rightarrow [\pi\ \omega]\varphi), \Delta & \text{(use case)}\end{array}}{\Gamma \vdash \{\mathcal{U}\}\,[\pi\ \texttt{while(}\ se\ \texttt{)}\ body\ \ \omega]\varphi, \Delta}$$

Loop invariant rules are based on an inductive argument: We have to prove that the invariant is *initially valid* and to show that it is *preserved* by an arbitrary iteration. Afterward, we may assume it for the execution of the remaining program $[\pi\ \omega]$ (*use case*). Since *preserved* and *use case* are to be proven in symbolic states where an arbitrary number of loop iterations has already been executed, potentially invalidating all information in the context, the context has to be *masked*. To this end, an "anonymizing" update $\mathcal{U}_{havoc}$ is added, which overwrites all variables/heap locations that are modified in the loop body with fresh symbols. In the context of simplistic programming languages, where only side-effect free expressions *se* are allowed for loop guards and there is no way of abruptly escaping the loop, this rule is already sufficient. For a language like Java, we need to take into account that loop guards might be complex expressions with side effects and exceptional behavior, and the execution might escape the loop in consequence of `return`s, `continue`s, `break`s, or thrown exceptions.

In the basic invariant rule loopInvariant, the loop body is executed outside its context $[\pi\ \omega]$. Consequently, information about how to handle `break`, `continue` and `return` statements is no longer present, and a direct extension of the rule that takes abrupt termination into account has to apply suitable program transformations to the loop body adding an encoding of this information. A fundamentally different approach based on four additional *labeled* modalities is discussed in detail in [22]; it requires five branches and is inherently incomplete. The approach implemented in the KeY system and described in [1] wraps the loop body in a labeled `try-catch` statement; `break`s, `return`s and `continue`s are transformed into labeled `break`s before which corresponding flags are set that describe the respective nature of the loop termination. Thrown exceptions

are caught in the catch block and assigned to a new variable which makes the exception available in the post condition of the *preserved* branch. An example for this transformation is given later in Example 3. The resulting invariant rule has the following form (the loop guard is executed twice in the preserved and use case branches since it may have side effects):

loopInvTransform

$$\frac{\begin{array}{ll}\Gamma \vdash \{\mathcal{U}\}\, Inv, \Delta & \text{(initially valid)}\\ \Gamma \vdash \{\mathcal{U}\}\, \{\mathcal{U}_{havoc}\}((Inv \wedge [\texttt{b=}nse]\texttt{b} \doteq \text{TRUE}) \rightarrow \widehat{[\texttt{b=}nse\texttt{;}\ body]Inv}), \Delta & \text{(preserved)}\\ \Gamma \vdash \{\mathcal{U}\}\, \{\mathcal{U}_{havoc}\}((Inv \wedge [\texttt{b=}nse]\texttt{b} \doteq \text{FALSE}) \rightarrow [\pi\ \texttt{b=}nse\texttt{;}\ \omega]\varphi), \Delta & \text{(use case)}\end{array}}{\Gamma \vdash \{\mathcal{U}\}\, [\pi\ \texttt{while(}\ nse\ \texttt{)\ \{}\ body\ \texttt{\}}\ \omega]\varphi, \Delta}$$

Here, $\widehat{\texttt{b=}nse\texttt{;}\ body}$ is the result of the mentioned program transformation, where Boolean flags `brk` and `rtrn` indicate that the loop has been left by a `break` or `return` statement, and the exception variable `exc` stores a thrown exception. The post condition $\widehat{Inv}$ of the *preserved* case has the following shape:

$$\begin{array}{ll}(\texttt{exc} \not\doteq null \rightarrow & [\pi\ \texttt{throw exc};\ \omega]\varphi)\\ \wedge\,(\texttt{brk} \doteq TRUE \rightarrow & [\pi\ \omega]\varphi)\\ \wedge\,(\texttt{rtrn} \doteq TRUE \rightarrow & [\pi\ \texttt{return result};\ \omega]\varphi)\\ \wedge\,(normal \rightarrow & Inv)\end{array}$$

where *normal* is equivalent to $\texttt{brk} \doteq FALSE \wedge \texttt{rtrn} \doteq FALSE \wedge \texttt{exc} \doteq null$. The special variable `result` is assigned the returned values in the transformed loop body. This approach has several drawbacks:

Exceptions in guards While loopInvTransform allows the loop guard *nse* to have side effects, it may not terminate abruptly. Relaxing this restriction introduces additional complexity.

Multiple reasons for loop termination In practice, there might be multiple reasons for abrupt loop termination. For instance, while attempting to return an expression including a division by zero, an exception will be thrown which ultimately causes the loop termination. In Java, the "return attempt" as a reason for the loop termination will be completely forgotten; when using the above invariant rule, however, two of the conjuncts in $\widehat{Inv}$ apply.

Understandability Due to the applied program transformation, the generated proof sequents are harder to understand for a human user, and also harder to describe in theory. Furthermore, the *preserved* case may also include the necessity to show the post condition φ. This may be considered as counter-intuitive since it is, theoretically, in the responsibility of the *use case*.

Repeated evaluation of loop guard The loop guard has to be evaluated four times according to the rule. This may constitute a performance problem in the verification process, since the guard might be a complex expression including, for instance, method calls and array accesses.

Moreover, program transformation of Java code is generally an intricate and error-prone task. Subsequently, we introduce a new syntactical entity called *loop*

scope. Loop scopes constitute a program abstraction which "sandboxes" loop bodies, thus facilitating a modular analysis of loops requiring very little program transformation. This new concept gives rise to a new kind of loop invariant rule.

Our proposed invariant rule is based on *(indexed) loop scopes* [22]. Definition 1 establishes *loop scope statements* as an extension to Java. We loosen the usual restriction that the label of a `continue` statement has to directly refer to a loop to allow for pushing leading loop labels inside loop scopes.

Definition 1 (Loop Scope Statements). *Let* `x` *be a program variable of type* `boolean`*, and p be a Java program. A* loop scope statement *is a Java statement of the form* $\circlearrowright_{\texttt{x}}\ p\ \circlearrowleft$*. We call* `x` *the* index *variable of the loop scope and p its* body*. Inside p, we allow labeled* `continue` *statements referring to arbitrary Java blocks.*

Definition 2 provides a scoping notion for `continue`s in loop scopes, which is needed for defining the semantics in Definition 3.

Definition 2 (Scope of Loop Scope Statements). *Let p be the body of a loop scope statement lst. A* `continue` *statement inside p is in the* scope *of lst iff it occurs on the top level, i.e., not nested inside a loop or loop scope in p. A labeled* `continue` *statement* `continue` *l is in the scope of lst iff the label l (1) is declared inside p and (2) refers to a top-level block in p.*

Definition 3 (Semantics of Loop Scope Statements). *Let lst be a loop scope statement with index* `x` *and body p. lst is* exited *by* `throw` *and (labeled)* `continue` *and* `break` *statements that are not caught by an inner* `catch` *or loop (scope) statement, or if there is no remaining statement to execute. Its semantics coincides with the semantics of p, except that upon exiting the loop scope,* `x` *is updated to (1)* `false` *if the exit point is a labeled or unlabeled* `continue` *statement in the scope of lst, and to (2)* `true` *for all other exit points. Furthermore, exiting the loop scope with* `x == false` *also leads to exiting the whole program.*

Example 2. Consider the program

`try {` $\circlearrowright_{\texttt{x}}$ `l:{ y+=2; continue l; f(); } ` $\circlearrowleft$ `} finally {y=0;}`

Following Definition 3, it is semantically equivalent to "`y+=2; x=false;`", since `y+=2;` does not exit the loop scope and the `continue` statement is in its scope.

We use the semantics of loop scopes (i.e., `x` is `false`, or *FALSE* in JavaDL, iff the loop continues with another iteration) to distinguish the *preserved* and the *use case* part in the second branch of our rule loopScopeInvariant (Fig. 1, next page), which subsumes the respective branches of loopInvTransform. The rule can be extended to a version for total correctness by reasoning about a well-founded relation [1]. Here, `x` is a fresh program variable, $\mathcal{U}_{havoc}$ an anonymizing update, and $n \geq 0$ is the number of labels in front of the loop. The program transformation performed by the rule is minimal. We merely (1) transform the `while` to an `if`, (2) push any labels inside, (3) add a trailing `continue` after the

$$\frac{\Gamma \vdash \{\mathcal{U}\}Inv, \Delta \qquad \Gamma, \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}Inv \vdash \Delta, \{\mathcal{U}\}\,\{\mathcal{U}_{havoc}\}[\pi\ \circlearrowright_{\mathtt{x}}\ \texttt{if (}nse\texttt{)}\ l_1 : \ldots l_n : \texttt{\{}\ body\ \texttt{continue;}\ \texttt{\}}\ \circlearrowleft\ \omega]((\mathtt{x} \doteq TRUE \rightarrow \varphi)\ \wedge\ (\mathtt{x} \doteq FALSE \rightarrow Inv))}{\Gamma \vdash \{\mathcal{U}\}[\pi\ \ l_1 : \ldots l_n : \texttt{while (}nse\texttt{) \{}\ body\ \texttt{\}}\ \omega]\varphi, \Delta}\ \textsf{loopScopeInvariant}$$

(initially valid)

(preserved & use case)

Fig. 1: The Loop Scope Invariant Rule

loop body, and (4) wrap the resulting `if` statement in a loop scope. Appending the `continue` statement ensures that the active statement of all final states arising after the execution of *body* is either a (labeled) `break` or `continue`, or a `throw` or `return` statement. The typical case where the loop scope has an empty body is the one that never entered the if statement, which corresponds to the case of regular loop termination due to an unsatisfied loop guard – the classic "use case" (the only other case is a labeled `break` referring to a label pointing to the loop). The following theorem states the validity of the rule loopScopeInvariant.

Theorem 1. *The rule* loopScopeInvariant *is* sound*, i.e., if the "initially valid" and "preserved & use case" premises are valid, then also the conclusion is valid.*

Proof sketch. The proof follows the usual inductive argument: The invariant has to hold upon entering the loop (ensured by the validity of the "initially valid" case) and after an arbitrary loop iteration. The latter is asserted by the semantics of the loop scope along with the addition of the `continue` statement and the post condition conjunct $\mathtt{x} \doteq FALSE \rightarrow Inv$: Since the second premise of the rule is valid, we know that whenever the loop is resuming with another iteration (and $\mathtt{x} \doteq FALSE$), the invariant is preserved. Furthermore, for the cases that the loop is exited, it holds that $\mathtt{x} \doteq TRUE$ and thus that the conclusion φ of $\mathtt{x} \doteq TRUE \rightarrow \varphi$, the post condition of the method, is true. Therefore, we can conclude the validity of the rule's conclusion.

Example 3. Fig. 2 depicts a synthetic example of a while loop with non-standard control flow taken from [1], as well as the "preserved" branch for the invariant rule loopInvTransform and the "preserved & used" branch for loopScopeInvariant, applied on the sequent $\Gamma \vdash \{\mathcal{U}\}\,[\texttt{while (x>=0) \{...\}}], \Delta$. Not only is the outcome for loopScopeInvariant already shorter and easier to read, but it also subsumes the "use case" branch of loopInvTransform which is not contained in Fig. 2. Also, the context $\pi\ \omega$ can constitute a Java program of arbitrary length. Since it occurs inside the additional modalities of the post condition in the "preserves" branch of loopInvTransform, this can significantly blow up the resulting sequent and therefore render the sequent even harder to understand. We additionally emphasize that loopScopeInvariant is easier to realize in systems like Hoare logic that do not allow more than one modality, which is required by loopInvTransform.

Of course, we need to treat loop scope statements in a sound manner according to their semantics (Definition 3). There are eight cases which we have

```
while (x >= 0) {
  if(x == 0) break;
  if(x == 1) return 42;
  if(x == 2) continue;
  if(x == 3) throw e;
  if(x == 4) x = -1;
}
```

(a) Loop with non-standard control flow.

$\Gamma, \{\mathcal{U}'\} Inv \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}($
$[\pi$ `boolean ls;` $\circlearrowleft_{\texttt{ls}}$
`if (x >= 0) {`
`  if(x == 0) {`
`    break;`
`  } if(x == 1) {`
`    return 42;`
`  } if(x == 2) {`
`    continue;`
`  } if(x == 3) {`
`    throw e;`
`  } if(x == 4) { x = -1; }`
`  continue;`
`}` $\circlearrowleft \omega]((\texttt{ls} \doteq TRUE \rightarrow \varphi) \wedge$
$(\texttt{ls} \doteq FALSE \rightarrow Inv))), \Delta$

(b) "preserved & used" branch of an application of loopScopeInvariant.

$\Gamma \vdash \{\mathcal{U}\}\{\mathcal{U}_{havoc}\}(Inv \wedge [\texttt{b=x>=0}]\texttt{b} \doteq TRUE \rightarrow$
`[b=x>=0;`
`loopBody: {`
`  try {`
`    boolean brk=false, rtrn=false;`
`    Throwable exc=null;`
`    if(x == 0) {`
`      brk=true; break loopBody;`
`    } if(x == 1) {`
`      result=42; rtrn=true;`
`      break loopBody;`
`    } if(x == 2) {`
`      break loopBody;`
`    } if(x == 3) {`
`      throw e;`
`    } if(x == 4) { x = -1; }`
`  } catch(Throwable e) {`
`    exc = e;`
`  }`
`  ]` $(\texttt{exc} \neq null \rightarrow [\pi\ \texttt{throw exc;}\ \omega]\varphi)$
$\wedge (\texttt{brk} \doteq TRUE \rightarrow [\pi\ \omega]\varphi)$
$\wedge (\texttt{rtrn} \doteq TRUE \rightarrow [\pi\ \texttt{return result;}\ \omega]\varphi)$
$\wedge (\texttt{exc} \doteq null \wedge \texttt{brk} \doteq FALSE \wedge$
$\texttt{rtrn} \doteq FALSE \rightarrow Inv)$
$), \Delta$

(c) "preserved" branch of an application of loopInvTransform.

Fig. 2: While loop with non-standard control flow and resulting sequents after an application of loopInvTransform and loopScopeInvariant on it.

to consider; those are distinguished by the currently active statement inside the loop scope, which can be: (1) empty, (2) an unlabeled `continue`, (3) a labeled `continue`, (4) an unlabeled `break`, (5) a labeled `break`, (6) a `return` for a void method, (7) a `return` for a non-void method, or (8) a `throw` statement. Fig. 3 shows those new calculus rules. We discuss the most interesting cases; for a full account as well as for the additionally relevant, already existing rules, see [19].

The rules labeledBreakIndexedLoopScope and labeledContinueIndexedLoopScope address the cases where a labeled `break` or `continue` reaches the loop scope. This only ever happens if the label is not addressing the current loop (or, for that matter, any block or inner loop inside the current loop): Otherwise, the already existing calculus rules of KeY will eventually transform the labeled to an unlabeled statement. If one of the two rules is applicable, the loop is definitely exited (and thus, the loop scope removed and `x` set to `true`), and the labeled `break` or `continue` statement is left for further processing outside this loop scope.

Due to the loop scope semantics (Definition 3), an unlabeled active `continue` statement has to trigger a leaving of the loop scope (removing the execution context $\pi\ \omega$) and the setting of the index variable to `false`. This is realized by the rule continueIndexedLoopScope, which distinguishes it from all the others that keep the context and set the index to `true`. It is applied either when the

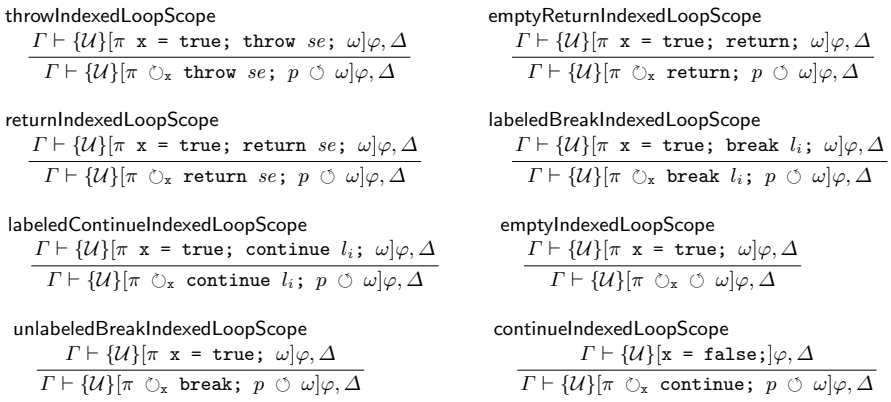


Fig. 3: Calculus rules for loop scope removal

additional `continue` statement added after the loop body is reached, i.e. in the case of normal control flow, or in the case of an (unlabeled, or labeled and referring to the current loop) `continue` statement within the loop body.

4 Evaluation

We implemented the loop scope invariant rule for KeY, a deductive program verification system for JavaDL based on heavyweight symbolic execution. Its calculus rules for the SE of Java programs cover most sequential Java features, such as inheritance, dynamic dispatch, reference types, recursive methods, exceptions, and strings (we refer the reader to [1] for a full account). Prior to that, the system was based on the loop invariant rule loopInvTransform (see Sect. 3). In the remainder of this section, we refer to the previous rule implemented in KeY as the "old" rule and to our implementation of the loop scope invariant rule as the "new" rule. Our experimental evaluation is based upon a sample of 54 Java programs (containing loops) of varying size which are shipped with KeY as examples. Each of these examples can be solved fully automatically by KeY. For the evaluation, we created two proof versions: One with the old, and one with the new rule. We then compared the numbers of proof nodes and SE steps for each example. Table 1 depicts the results for 44 of the examples. Negative numbers indicate a better performance of the new rule. We left out some small examples for space reasons; the complete table and the KeY proof files can be found on our web page `key-project.org/papers/loopscopes`.

Fig. 4 contains box plots for the percentage difference of the numbers of proof nodes and SE steps between the old and the new rule. The bars in the middle of the box represent the median, the box itself the midspread (the middle 50%), and the whiskers point to the last items that are still within 1.5 of the inter quartile range of the lower/upper quartile. The examples which are not covered by the whiskers, the outliers, are signified as points.

Overall, we saved between 3% and 63% of *SE steps*, the median is 27. Of all examples, 50% are in the range of 17% and 32% of saved steps. This is mostly

due to the overhead of the fourfold evaluation of loop guards in the old rule. Considering the total number of saved *proof steps*, the situation is more complex. While for 50% of all examples, the number of proof steps can be reduced by 7% to 16% when using the new rule, we have seven outliers, and in total four examples where the number of nodes is higher in the proof with the new rule.

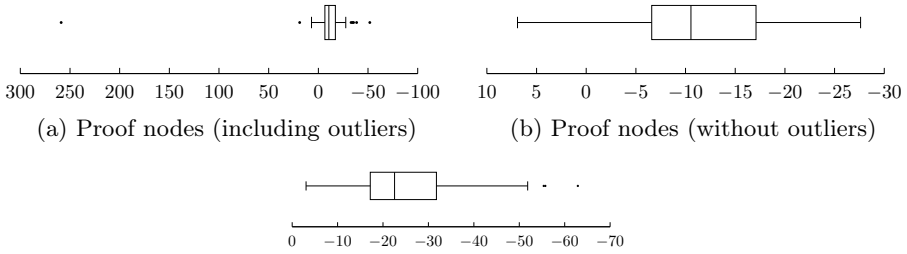
Of those, the "coincidence_count" example with an increase of 258.88% is most surprising. The reason is not the SE, since even in this example we saved 27.14% of SE steps. We discovered that the increased number of proof nodes is due to disadvantageous decisions of KeY's strategies: From situations in the compared proofs where the sequents were equal up to renaming of constants and ordering of formulas, the strategies made significantly worse decisions in the proof with the new rule. We made similar observations for the remaining three negative examples as well as for the positive outlier "jml-information-flow". Exemplarily for "coincidence_count" and "jml-information-flow", we were able to underpin the assumption that the extreme loss/gain in performance is due to (fixable) disadvantageous strategy decisions by pruning the longer proofs at the interesting positions, performing a few simple steps manually and starting the strategies again. The resulting proof size savings fit the expectations. We reported those examples to the KeY team as benchmarks for tuning the strategies.

Some of the positive outliers are more interesting: In the "lcp" example, the loop condition is extremely complex, which is why the new rule performs much better. The "ArrayList.remove.0" example contains two nested loops. The application of an invariant rule to the inner loop is superfluous, since the specific method contract constituting the proof goal already facilitates closing the proof without considering the inner loop. Still, the strategies choose to apply an invariant rule. While in the case of the new rule, this is not very costly and the proof can be closed without any further branching, the proof with the old rule spends a lot of proof steps for the use case of the inner loop.

5 Exploiting Invariants: Integration of State Merging

As mentioned in the introduction, one of the main bottlenecks of symbolic execution is the *path explosion problem* [5]. In [18], a general lattice-based framework for merging states in SE is proposed and implemented for KeY. SE states sharing the same program counter (the same remaining program to execute) can be merged together using one of the state merging techniques conforming with the framework. The most common techniques are *if-then-else merging*, where the precise values of differing program variables in the merged states are remembered and distinguished by the respective path conditions, and *predicate abstraction*.

The easiest (and automatic) state merging technique is the if-then-else method, which though only partially improves the situation, since at the end, if-then-else expressions will be split up again. Conversely, predicate abstraction is a strong technique, which though requires the user to supply abstraction predicates by JML annotations; the automatic generation of those predicates is, similar to loop invariant inference, a difficult task, and not yet implemented for KeY. However,



(a) Proof nodes (including outliers)

(b) Proof nodes (without outliers)

(c) Symbolic execution steps

Fig. 4: Box plots visualizing the percentage difference in the number of proof nodes / SE steps between the old and the new rule.

Problem	Proof Nodes		% Difference	Symb.Ex.Steps		% Difference
	Old Rule	New Rule	# Nodes	Old Rule	New Rule	# Symb.Ex.Steps
coincidence_count	14.199	50.957	258.88%	210	153	-27.14%
ArrayList.remove.1	12.269	14.575	18.80%	258	191	-25.97%
saddleback_search	30.119	32.203	6.92%	235	181	-22.98%
list_recursiveSpec	5.243	5.557	5.99%	184	170	-7.61%
removeDups	19.891	19.736	-0.78%	373	308	-17.43%
ArrayList_add	6.451	6.380	-1.10%	458	444	-3.06%
polishFlagSort	4.299	4.242	-1.33%	93	83	-10.75%
ArrayList_concatenate	23.205	22.585	-2.67%	641	564	-12.01%
list_recursiveSpec	6.131	5.937	-3.16%	216	184	-14.81%
BinarySearch_search	4.462	4.269	-4.33%	182	149	-18.13%
MemoryAllocator_alloc	1.067	1.003	-6.00%	90	77	-14.44%
reverseArray	5.348	4.997	-6.56%	151	139	-7.95%
Node_search	7.768	7.256	-6.59%	97	57	-41.24%
gcdHelp-post	2.634	2.456	-6.76%	39	28	-28.21%
Queens_isConsistent	3.677	3.420	-6.99%	167	135	-19.16%
ArrayList.enlarge	3.051	2.824	-7.44%	106	79	-25.47%
ArrayList.contains	2.414	2.225	-7.83%	98	60	-38.78%
UpdateAbstraction_ex9_secure	1.457	1.319	-9.47%	183	162	-11.48%
MemoryAllocator_alloc_unsigned	1.362	1.232	-9.54%	91	78	-14.29%
ArrayList_enlarge	2.764	2.499	-9.59%	152	125	-17.76%
arrayMax	1.921	1.734	-9.73%	97	72	-25.77%
arrayFillNonAtomic	5.376	4.852	-9.75%	294	268	-8.84%
ArrayList_enlarge	3.195	2.871	-10.14%	157	130	-17.20%
SumAndMax_sumAndMax	4.101	3.676	-10.36%	140	114	-18.57%
ArrayList.add	2.302	2.060	-10.51%	144	131	-9.03%
LinkedList_get_normal	6.889	6.160	-10.58%	184	159	-13.59%
removeDups_arrayPart	1.735	1.533	-11.64%	102	89	-12.75%
reverseArray2	2.224	1.964	-11.69%	134	110	-17.91%
selection_sort	5.512	4.829	-12.39%	278	205	-26.26%
ArrayList.remFirst	2.485	2.175	-12.47%	168	133	-20.83%
loop2	1.032	892	-13.57%	83	57	-31.33%
AddAndMultiply_add	1.351	1.165	-13.77%	109	83	-23.85%
permissions_method3	1.656	1.401	-15.40%	91	57	-37.36%
contains	1.021	863	-15.48%	73	49	-32.88%
project	6.137	5.088	-17.09%	433	293	-32.33%
for_Array	827	684	-17.29%	95	68	-28.42%
ArrayList_get	1.830	1.496	-18.25%	157	121	-22.93%
sum1	939	753	-19.81%	85	58	-31.76%
sum3	820	646	-21.22%	100	58	-42.00%
ArrayList_contains_dep	6.069	4.393	-27.62%	396	213	-46.21%
ArrayList.remove.0	3.689	2.473	-32.96%	186	69	-62.90%
jml-information-flow	48.215	31.659	-34.34%	474	369	-22.15%
lcp	3.132	1.927	-38.47%	235	104	-55.74%
for_Iterable	622	300	-51.77%	130	58	-55.38%

Table 1: 44 out of 54 experimental results (including all negative results), ordered by the percentage of proof nodes saved. The outliers are discussed in Sect. 4.

```
/*@ public normal_behavior
  @ requires arr != null;
  @ ensures \result == -1 ||
  @       arr[\result] == elem;
  @*/
public int partiallyUnrolledFindBrk(
    int[] arr, int elem) {
  int i = -1, res = -1;
  /*@ loop_invariant
    @    (\forall int k; k <= i && k >= 0;
    @          arr[k] != elem) &&
    @    i >= -1 && i <= arr.length &&
    @    (res == -1 || arr[res] == elem);
    @ decreases arr.length - i + 1;
    @*/
  while (++i < arr.length) {
    if (i + 3 < arr.length) {
      if (arr[i] == elem) {
        res = i; break;
      } else if (arr[i + 1] == elem) {
        res = i + 1; break;
      } else if (arr[i + 2] == elem) {
        res = i + 2; break;
      } else if (arr[i + 3] == elem) {
        res = i + 3; break;
      } else {
        i += 3; continue;
      }
    }

    if (arr[i] == elem)
      res = i; break;
  }

  return res;
}
```

Listing 1: Find method using `break` statements to escape the loop

```
/*@ public normal_behavior
  @ requires arr != null;
  @ ensures \result == -1 ||
  @       arr[\result] == elem;
  @*/
public int partiallyUnrolledFindRtrn(
    int[] arr, int elem) {
  int i = -1;
  /*@ loop_invariant
    @    (\forall int k; k <= i && k >= 0;
    @          arr[k] != elem) &&
    @    i >= -1 && i <= arr.length;
    @
    @ decreases arr.length - i + 1;
    @*/
  while (++i < arr.length) {
    if (i + 3 < arr.length) {
      if (arr[i] == elem) {
        return i;
      } else if (arr[i + 1] == elem) {
        return i + 1;
      } else if (arr[i + 2] == elem) {
        return i + 2;
      } else if (arr[i + 3] == elem) {
        return i + 3;
      } else {
        i += 3; continue;
      }
    }

    if (arr[i] == elem)
      return i;
  }

  return -1;
}
```

Listing 2: Find method using `return` statements to directly return the result

when merging states resulting from the execution of loops with abrupt termination (and not arbitrary states, e.g., resulting from a split after an if statement), we can *automatically* exploit the loop invariant as well as the post condition for the method to generate suitable abstraction predicates that can be employed for predicate abstraction. Based on [18], we implemented this approach for KeY (available at `key-project.org/papers/loopscopes`). When applying our loop invariant rule, the appropriate merge points and inferred abstraction predicates are registered and taken into account by the automatic strategies. Once all execution paths until a merge point are explored, they are merged based on this information. We describe how to infer the predicates along an example.

Listings 1 and 2 contain similar, "partially unrolled" methods for finding an element in an integer array. The methods are fully specified in JML and can be proven by KeY. As long as possible, they search the next three array positions for the sought-after element. In Listing 1, the control flow breaks out of the loop once that the element is found; in Listing 2, the element is directly returned. SE produces proof goals for each `break`/`return` statement, which can be merged.

In Listing 1, the states after each break only differ in the value of the variable `res`, since `i` is not needed anymore after the loop and is removed. For each state,

the part of the invariant talking about `res` has to hold: `res == -1 || arr[res] == elem`. From this formula, we create a unary abstraction predicate $P_{break}(v) \equiv v \doteq -1 \lor \texttt{arr}[v] \doteq \texttt{elem}$. KeY is able to show in a background proof that this predicate holds for `res` in each state and uses it to abstract away from the concrete values in the merged state. Thus, we save 194 proof nodes (6.3%) and 23 symbolic execution steps (11.6%). Compared to using the *old* invariant rule, we save 21.0% / 45.8% of proof nodes / symbolic execution steps.

For Listing 2, we can do something similar based on the post condition of the method. The states after the return statements differ in the returned value. We generate an abstraction predicate from the post condition of the method by substituting the JML expression `\result` by the parameter of the predicate: $P_{return}(v) \equiv v \doteq -1 \lor \texttt{arr}[v] \doteq \texttt{elem}$. The obvious equivalence of P_{break} and P_{return} is due to fact that (almost) the method's whole behavior is realized in the loop. KeY proves this property true for each returned value in the return states and merges the states based on the abstraction predicate. We obtain a reduction of 164 proof nodes (6.2%) and 20 symbolic execution steps (10.0%); and 10.5% / 31.7% compared to the old invariant rule.

6 Related Work

It is natural to compare our work with other heavyweight SE systems like VeriFast and KIV. For VeriFast, an SE system for C, we unfortunately could not find any work formally explaining the handling of irregular control flow in loops; the most formal paper we encountered [21] is based on a reduced language without `throw`s, `break`s and `continue`s. KIV is a deductive verification system which has been extended by an SE calculus covering Java Card in a PhD thesis by Stenzel [20]. Their calculus is also a variant of Dynamic Logic. Its most significant difference to JavaDL is the *flattening* (sequential decomposition) of statements. This implies that the system cannot use inactive prefixes, but instead includes *mode information* in a store shared by multiple modalities, and multiple artificial statements dealing with method returns and abrupt termination. Interestingly, their loop invariant rule bears a strong resemblance to the one proposed by us. Where we decide whether to prove the invariant or the "use case" based on the loop scope index, they decide based on the evaluation of the loop guard and on the mode information. But there are some relevant aspects which distinguish this work from ours: (1) The rule in KIV requires substantially more program transformation due to the flattening. Moreover, we can directly treat `continue` statements, whereas they are transformed to labeled `break`s in KIV. One of their arguments is that `continue`s are problematic for loop unwinding; however, as discussed in [22], loop scopes can also be employed in that context, making the transformation superfluous. (2) In [20], the rule circumvents the need for anonymization by dropping the preconditions Γ, which makes it necessary to also encode information about the initial state in the invariant, thus bloating it more than necessary. (3) After an abrupt termination, KIV has to process all subsequent modalities until an appropriate "catcher" statement appears. Our

approach simply exits the loop scope, which emphasizes the advantages of the "sandboxing" technique. (4) Our work is, to the best of our knowledge, the only one comparing the performance of a "classic" invariant rule to one of this style, and the only one integrating an invariant rule with symbolic state merging. Current versions of KIV can no longer parse Java programs, hence it was not possible to practically examine the implemented rule.

A lot of work on the verification of sequential programs is based on Verification Condition Generation (VCG). ESC/Java(2) [10] and its successor OpenJML [6] generate verification conditions for annotated Java programs. The Frama-C plugins Jessie and Krakatoa [15] translate annotated C and Java programs into the Why [3] language. Boogie [2] generates verification conditions for Spec#. In these approaches, the verification works via a translation to an intermediate language. The way loops are commonly translated ("loop framing", [2]) is structurally similar to our approach: The invariant is asserted initially, accessed locations are anonymized and the invariant is assumed for the anonymized state; finally, the invariant is asserted after executing the loop body. The handling of abnormal control flow depends on the translation into the intermediate language; usually, this remains rather underspecified in the literature. According to a personal communication with David R. Cok, exceptions in OpenJML result in `goto`s to basic blocks for `catch` statements or exceptional exit from the procedure; `break`s and `continue`s likewise branch to dedicated blocks. Generally, verification conditions consist of one huge implication per method, including one conjunct for each program block ending in a `goto`. While probably being beneficial for the performance of VCG approaches, this impedes the traceability of problems. Conversely, Symbolic Execution (SE) produces many small proof obligations. Our approach targets a middle course. It is based on SE, but reduces the number of proof goals through abstraction-based state merging, while increasing understandability by using a loop invariant rule with a simple semantics. Additionally, we require very little program transformation. The translation into an intermediate language may mitigate language complexity; however, it can require compromises concerning soundness [10] and, in any case, is a non-trivial and error-prone task [15] which is difficult to prove sound.

7 Future Work and Conclusion

We have introduced the concept of a *loop scope* for the deduction-based symbolic execution of loops in industrial sequential programming languages. Building on this, we have presented a loop invariant rule which we implemented for the program verification system KeY. Our rule is sound, efficient, and produces understandable proof obligations. We integrated the new rule with a novel, fully automatic abstraction-based state merging technique based on abstraction predicates inferred from existing loop invariants and method post conditions. The performance improvement is beneficial for automatic proof attempts, where thresholds on time or number of proof steps may otherwise lead to early termination.

The loop scope invariant rule is scheduled to replace the existing rule in KeY in the next public release. We are planning to also release our state merging approach to the public after having performed a more extensive case study.

References

1. Ahrendt, W., Beckert, B., et al. (eds.): Deductive Software Verification – The KeY Book, LNCS, vol. 10001. Springer International Publishing (2016)
2. Barnett, M., Chang, B.Y.E., et al.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: Intern. Symp. on FMCO. pp. 364–387. Springer (2005)
3. Bobot, F., Filliâtre, J.C., et al.: Why3: Shepherd Your Herd of Provers. In: Boogie 2011: First International Workshop on IVL. pp. 53–64 (2011)
4. Cadar, C., Dunbar, D., et al.: KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In: 8th USENIX Conference on OSDI. pp. 209–224. USENIX Association, Berkeley, CA, USA (2008)
5. Cadar, C., Sen, K.: Symbolic Execution for Software Testing: Three Decades Later. Communications of the ACM 56(2), 82–90 (2013)
6. Cok, D.R.: OpenJML: Software Verification for Java 7 Using JML, OpenJDK, and Eclipse. In: Proceedings 1st Workshop on FIDE. pp. 79–92 (2014)
7. Cuoq, P., Kirchner, F., et al.: Frama-C. In: SEFM'12. pp. 233–247. Springer (2012)
8. Dannenberg, R., Ernst, G.: Formal Program Verification Using Symbolic Execution. IEEE Transactions on Software Engineering SE-8(1), 43–52 (1982)
9. Filliâtre, J.C.: Deductive Software Verification. International Journal on Software Tools for Technology Transfer (STTT) 13(5), 397–403 (2011)
10. Flanagan, C., Saxe, J.B.: Avoiding Exponential Explosion: Generating Compact Verification Conditions. SIGPLAN Not. 36(3), 193–205 (Jan 2001)
11. Gouw, S.d., Rot, J., et al.: OpenJDK's java.utils.Collection.sort() is broken: The good, the bad and the worst case. In: Kroening, D., Pasareanu, C.S. (eds.) Proc. of the 27th Intl. Conf. on Computer Aided Verification. Springer (2015)
12. Hentschel, M., Hähnle, R., et al.: Visualizing Unbounded Symbolic Execution. In: Seidl, M., Tillmann, N. (eds.) Tests and Proofs, pp. 82–98. LNCS, Springer (2014)
13. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. Communications of the ACM 12(10), 576–580 (1969)
14. Jaffar, J., Murali, V., et al.: Boosting Concolic Testing via Interpolation. In: Proceedings of 9th Joint Meeting on FSE. pp. 48–58. ACM, New York, USA (2013)
15. Marché, C., Paulin-Mohring, C., et al.: The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs Annotated in JML. The Journal of Logic and Algebraic Programming 58(1–2), 89 – 106 (2004)
16. Pariente, D., Ledinot, E.: Formal Verification of Industrial C Code using Frama-C: A Case Study. Proc. of the 1st Intl. Conf. on FoVeOOS p. 205 (2010)
17. Păsăreanu, C.S., Visser, W.: Verification of Java Programs Using Symbolic Execution and Invariant Generation. In: Graf, S., Mounier, L. (eds.) Model Checking Software, pp. 164–181. Springer Berlin Heidelberg (2004)
18. Scheurer, D., Hähnle, R., et al.: A General Lattice Model for Merging Symbolic Execution Branches. In: Ogata, K., Lawford, M., et al. (eds.) ICFEM 2016, Proceedings. pp. 57–73. Springer International Publishing (2016)
19. Steinhöfel, D., Wasser, N.: A New Invariant Rule for the Analysis of Loops with Non-standard Control Flows. Tech. rep., TU Darmstadt (2017), `http://tinyurl.com/loop-scopes-tr`

20. Stenzel, K.: Verification of Java Card Programs. Ph.D. thesis, University of Augsburg, Germany (2005)
21. Vogels, F., Jacobs, B., et al.: Featherweight VeriFast. LMCS 11(3) (2015)
22. Wasser, N.: Automatic Generation of Specifications using Verification Tools. Ph.D. thesis, Technische Universität Darmstadt, Darmstadt (January 2016)