Diagnostics in Probabilistic Program Verification

Dafny 2026 at POPL – Rennes, France

Philipp Schröer, **Darion Haase**, Joost-Pieter Katoen

RWTH Aachen University

January 11, 2026

Formally Justified Diagnostics for Verification

What if verification fails?

- *What* is the issue?
- *Which* statement(s) are the ’cause’?

```
method DutchFlag(a: array<Color>)
  requires a != null modifies a
  ensures forall i,j :: 0 <= i < j < a.Length  ==>  Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w != b
    invariant 0 <= r <= w <= b <= a.Length;
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]))
  {   match a[w]
        case Red =>
          a[r], a[w] := a[w], a[r];
          r, w := r + 1, w + 1;
        case White =>
          w := w + 1;
        case Blue =>
          b := b - 1;
          a[w], a[b] := a[b], a[w];
  }
}
```

What if verification fails?

- *What* is the issue?
- *Which* statement(s) are the ’cause’?

What if verification succeeds?

- *Why* did verification succeed?
- *Which* statement(s) are ’relevant’?

```
method DutchFlag(a: array<Color>)
  requires a != null modifies a
  ensures forall i,j :: 0 <= i < j < a.Length  ==>  Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w != b
    invariant 0 <= r <= w <= b <= a.Length;
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]))
  {   match a[w]
        case Red =>
          a[r], a[w] := a[w], a[r];
          r, w := r + 1, w + 1;
        case White =>
          w := w + 1;
        case Blue =>
          b := b - 1;
          a[w], a[b] := a[b], a[w];
  }
}
```

```
method DutchFlag(a: array<Color>)
  requires a != null modifies a
  ensures forall i,j :: 0 <= i < j < a.Length  ==>  Ordered(a[i], a[j])
  ensures multiset(a[..]) == old(multiset(a[..]))
{
  var r, w, b := 0, 0, a.Length;
  while w != b
    invariant 0 <= r <= w <= b <= a.Length;
    invariant forall i :: 0 <= i < r ==> a[i] == Red
    invariant multiset(a[..]) == old(multiset(a[..]))
  {   match a[w]
        case Red =>
          a[r], a[w] := a[w], a[r];
          r, w := r + 1, w + 1;
        case White =>
          w := w + 1;
        case Blue =>
          b := b - 1;
          a[w], a[b] := a[b], a[w];
  }
}
```

What if verification fails?

- *What* is the issue?
- *Which* statement(s) are the 'cause'?

What if verification succeeds?

- *Why* did verification succeed?
- *Which* statement(s) are 'relevant'?

▷ Improve the *verification loop*: attempt verification, failure, targeted refine.

▷ Good diagnostics are key for formal verification.

Caesar aims to be 'the probabilistic version of Dafny' (presented at Dafny 2024)

- Based on a *quantitative intermediate verification language*
    - also has verification statements like `assert`, `assume`, ...

Diagnostics for Caesar

Caesar aims to be 'the probabilistic version of Dafny' (presented at Dafny 2024)

- Based on a *quantitative intermediate verification language*
  - also has verification statements like `assert`, `assume`, ...

The Three Pillars of Caesar:

(1) **Probabilistic** programs with $\mathbb{R}^{\infty}_{\geq 0}$-valued specifications,

(2) **Quantitative verification statements** to encode a range of **proof rules**,

(3) **Duality:** reasoning about lower/upper bounds of expected values (`assert` and `coassert`, ...).

Caesar aims to be 'the probabilistic version of Dafny' (presented at Dafny 2024)

- Based on a *quantitative intermediate verification language*
  - also has verification statements like `assert`, `assume`, ...

The Three Pillars of Caesar:

(1) **Probabilistic** programs with $\mathbb{R}_{\geq 0}^{\infty}$-valued specifications,

(2) **Quantitative verification statements** to encode a range of **proof rules**,

(3) **Duality:** reasoning about lower/upper bounds of expected values (`assert` and `coassert`, ...).

▷ How to generalize error reporting/diagnostics to the probabilistic setting?

Slicing for Error Reporting

x := flip$\langle 1/2 \rangle$
$\texttt{assert}\ x \geq 0$
$\texttt{assert}\ x = 1$

Idea: Errors are caused by presence of statements that create unfulfillable proof obligations.

Slicing for Error Reporting

x := $\mathsf{flip}\langle 1/2 \rangle$
$\texttt{assert}\ x \geq 0$
$\texttt{assert}\ x = 1$

Idea: Errors are caused by presence of statements that create unfulfillable proof obligations.

```
x := flip⟨1/2⟩
assert x ≥ 0
assert x = 1
```

Idea: Errors are caused by presence of statements that create unfulfillable proof obligations.

Error localization. Error-witnessing Slice:
Subprogram S' of program S such that:

(1) S' has a counter-example, and

(2) $\forall \sigma.\ \sigma$ is a counter-example in $S' \implies \sigma$ is a counter-example in S

▷ Report an error at the remaining assertion(s).

Contributions

- Theory for diagnostics, suitable for *classical and probabilistic programs*:

  (1) **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),

Contributions

- Theory for diagnostics, suitable for *classical and probabilistic programs*:

  (1) **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),

  (2) **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),

Contributions

- Theory for diagnostics, suitable for *classical and probabilistic programs*:
  1. **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),
  2. **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),
  3. **Hints** by verification-preserving slices (tailoring program to specification).

- Theory for diagnostics, suitable for *classical and probabilistic programs*:
  1. **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),
  2. **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),
  3. **Hints** by verification-preserving slices (tailoring program to specification).
- Defined for *quantitative* intermediate verification language (IVL).
- Specialized diagnostics for proof rules (induction, procedure calls).
- Slicing engine *Brutus* implemented in *Caesar*.

Reason about loops by finding a suitable loop *invariant* I:

```
// pre: X
while b invariant I {
  C
}
// post: Y
```

$\rightsquigarrow$

```
// pre: X
assert I;
havoc variables;
validate;
assume I;
if b {
  C;
  assert I;
  assume false
} else { }
// post: Y
```

Minimal error-witnessing slice S' removing from $\{①, ②\}$:

```
// pre: X
① assert I ;
  havoc variables;
  validate;
  assume I;
  if b {
    C;
② assert I ;
    assume false
  } else { }
// post: Y
```

Minimal error-witnessing slice S' removing from $\{ ①, ② \}$:

```
    // pre: X
①   assert I;
    havoc variables;
    validate;
    assume I;
    if b {
      C;
②     assert I;
      assume false
    } else { }
    // post: Y
```

$① \in S' \implies$ **Pre does not entail invariant.**

$\exists \sigma.\ X(\sigma) \not\Rightarrow I(\sigma).$

Minimal error-witnessing slice S' removing from $\{ ①, ② \}$:

```
     // pre: X
①   assert I;
     havoc variables;
     validate;
     assume I;
     if b {
       C;
②     assert I;
       assume false
     } else { }
     // post: Y
```

① $\in S' \implies$ **Pre does not entail invariant.**
$\exists \sigma.\ X(\sigma) \not\Rightarrow I(\sigma)$.

② $\in S' \implies$ **Invariant not inductive.**
$\exists \sigma.\ \sigma \models b$ and $\sigma \not\models \{ I \}\ C\ \{ I \}$.

Minimal error-witnessing slice S' removing from $\{①, ②\}$:

```
    // pre: X
①  assert I;
    havoc variables;
    validate;
    assume I;
    if b {
      C;
②    assert I;
      assume false
    } else { }
    // post: Y
```

$① \in S' \implies$ **Pre does not entail invariant.**
$\exists \sigma.\ X(\sigma) \not\Rightarrow I(\sigma)$.

$② \in S' \implies$ **Invariant not inductive.**
$\exists \sigma.\ \sigma \models b$ and $\sigma \not\models \{ I \}\ C\ \{ I \}$.

$①, ② \notin S' \implies$ **Invariant does not entail post.**
$\exists \sigma.\ \sigma \not\models b$ and $I(\sigma) \not\Rightarrow Y(\sigma)$.

Assert-like (Reductive) Statements:

- Boolean intuition: can only *reduce the number of verifying paths*.
- Probabilistic: expected value of X after executing $S \leq X$.
- in Caesar: e.g. `assert`, `coassume`, `validate`, `havoc`.

Assert-like (Reductive) Statements:

- Boolean intuition: can only *reduce the number of verifying paths*.
- Probabilistic: expected value of X after executing $S \;\leq\; X$.
- in Caesar: e.g. `assert`, `coassume`, `validate`, `havoc`.

From Reductive Statements to Error-Witnessing Slice:

Erase reductive statements from S to obtain S'. Then,

$$S' \text{ has an error} \quad\Longrightarrow\quad S' \text{ is an error-witnessing slice of } S.$$

Recall *error-witnessing slice* S' of S: (1) S' has an error; and (2) errors in S' are also in S.

Certificates. **Verification-witnessing Slice:**

Subprogram S' of program S such that:

$$S' \text{ verifies} \implies S \text{ verifies.}$$

Certificates. **Verification-witnessing Slice:**

Subprogram S' of program S such that:

$$S' \text{ verifies} \implies S \text{ verifies.}$$

Assume-like (Extensive) Statements:

- Boolean intuition: can only *reduce the number of erroring paths*.
- Erase extensive statements from S to obtain verification-witnessing slice S'.

Certificates. **Verification-witnessing Slice:**

Subprogram S' of program S such that:

$$S' \text{ verifies} \implies S \text{ verifies.}$$

Assume-like (Extensive) Statements:

- Boolean intuition: can only *reduce the number of erroring paths*.
- Erase extensive statements from S to obtain verification-witnessing slice S'.

Hints. **Verification-preserving Slice:**

Subprogram S' of program S such that:

$$S \text{ verifies} \implies S' \text{ verifies.}$$

▷ Remove statements while preserving the specification.

Induction Proof Rule – From Slices to Hints

Verification-witnessing slice S' removing from $\{$ Ⓘ, Ⓘⓘ $\}$:

```
// pre: X
assert I;
havoc variables;
validate;
```
Ⓘ `assume I;`
```
if b {
  C;
  assert I;
```
Ⓘⓘ `  assume false`
```
} else { }
// post: Y
```

Verification-witnessing slice S' removing from $\{ ⓘ, ⓘⓘ \}$:

```
// pre: X
assert I;
havoc variables;
validate;
Ⓘ assume I;
if b {
  C;
  assert I;
ⒾⒾ assume false
} else { }
// post: Y
```

$ⓘ \notin S' \implies$ Assuming the invariant is not necessary.

$\models \{ X \}$ `while` b `invariant true` $\{C\}$ $\{ Y \}$

Verification-witnessing slice S' removing from $\{ \text{Ⓘ}, \text{Ⓘⓘ} \}$:

```
// pre: X
assert I;
havoc variables;
validate;
Ⓘ assume I;
if b {
  C;
  assert I;
Ⓘⓘ assume false
} else { }
// post: Y
```

Ⓘ $\notin S' \implies$ Assuming the invariant is not necessary.

$\models \{ X \}$ `while` b `invariant true` $\{C\} \{ Y \}$

Ⓘⓘ $\notin S' \implies$ While loop could be an if statement.

$\models \{ X \}$ `if` b $\{C\}$ `else` $\{\} \{ Y \}$

Verification-preserving slices tailor the program to the specification:

$$
\begin{array}{l}
\texttt{assume}\ x_{in} \geq 0\texttt{;} \\
x \;:=\; x_{in}\texttt{;} \\
\texttt{if}\ x < 0\ \{ \\
\quad x \;:=\; -x \\
\}\ \texttt{else}\ \{\ \}\texttt{;} \\
\texttt{assert}\ x = |x_{in}|
\end{array}
\qquad \checkmark
$$

Verification-preserving slices tailor the program to the specification:

```
assume x_in ≥ 0;
x := x_in;
if x < 0 {
  x := −x
} else { };
assert x = |x_in|
```

✓

Given program S and sliceable statements $S_1, \ldots, S_n$,

1. **Pre-processing:** Transform $S_i \rightsquigarrow$ `if` $enabled_{S_i}$ $\{S_i\}$ `else` $\{\}$

Given program S and sliceable statements $S_1, \dots, S_n$,

1. **Pre-processing:** Transform $S_i \rightsquigarrow \texttt{if}\ enabled_{S_i}\ \{S_i\}\ \texttt{else}\ \{\}$
2. **Solving for slices:**
   - Erroring slices: Solve SMT query

$$\exists enabled_{S_1}, \dots, enabled_{S_n}.\ \left(\exists \sigma \in \mathsf{States}.\ \mathsf{vp}[\![S]\!](\top)(\sigma) \neq \top\right)\ .$$

▷ We support *first* counter-example and *globally optimal* by size (binary search).

Given program S and sliceable statements $S_1, \ldots, S_n$,

1. **Pre-processing:** Transform $S_i \rightsquigarrow$ `if` $enabled_{S_i}$ $\{S_i\}$ `else` $\{\}$
2. **Solving for slices:**
   - **Erroring slices:** Solve SMT query

$$\exists enabled_{S_1}, \ldots, enabled_{S_n}.\ \left(\exists \sigma \in \text{States}.\ \text{vp}[\![S]\!](\top)(\sigma) \neq \top\right)\ .$$

     - ▷ We support *first* counter-example and *globally optimal* by size (binary search).

   - **Verifying slices:** Solve SMT query

$$\exists enabled_{S_1}, \ldots, enabled_{S_n}.\ \left(\forall \sigma \in \text{States}.\ \text{vp}[\![S]\!](\top)(\sigma) = \top\right)\ .$$

     - ▷ Quantifier alternation makes this more difficult.
     - ▷ Using *unsat cores*, also support locally/globally optimal slices (minimum UNSAT subset search).

Discussion & Evaluation

We tested our slicer *Brutus* on 46 benchmarks (probabilistic and Boolean), mostly quantifier-free.

[1] https://github.com/boogie-org/boogie/issues/1008

We tested our slicer *Brutus* on 46 benchmarks (probabilistic and Boolean), mostly quantifier-free.

Erroring slices:

- Error localization is done like in Boogie/Dafny (*first* counter-example)
  - Boogie's error localization is based on inconsistent models[1]
  - Caesar tries to keep SMT problem quantifier-free if possible
- Minimization is cheap and helps ($\leq +20$ ms, from ≤ 2 ms for 80% of benchmarks)

[1] `https://github.com/boogie-org/boogie/issues/1008`

We tested our slicer *Brutus* on 46 benchmarks (probabilistic and Boolean), mostly quantifier-free.

Erroring slices:

- Error localization is done like in Boogie/Dafny (*first* counter-example)
  - Boogie's error localization is based on inconsistent models[1]
  - Caesar tries to keep SMT problem quantifier-free if possible
- Minimization is cheap and helps ($\leq +20$ ms, from ≤ 2 ms for 80% of benchmarks)

Verifying slices:

- UNSAT core is cheap, often suboptimal. Minimization more expensive but helps.
- Dafny's *verification coverage report* reports statements necessary for verification
  $\Rightarrow$ corresponds to *verification-witnessing slices*?

[1] https://github.com/boogie-org/boogie/issues/1008

Summary

- Theory for diagnostics, suitable for *classical and probabilistic programs*:
  1. **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),
  2. **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),
  3. **Hints** by verification-preserving slices (tailoring program to specification).
- For *quantitative* intermediate verification language (IVL), and with specialized diagnostics.
- Implemented in Caesar tool; demonstrated on benchmarks.

Summary

- Theory for diagnostics, suitable for *classical and probabilistic programs*:
  1. **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),
  2. **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),
  3. **Hints** by verification-preserving slices (tailoring program to specification).
- For *quantitative* intermediate verification language (IVL), and with specialized diagnostics.
- Implemented in Caesar tool; demonstrated on benchmarks.

Paper accepted at *ESOP 2026* (preprint online): *“Error Localization, Certificates, and Hints for Probabilistic Program Verification via Slicing”*.

- Upper bounds (e.g. `coassert`), specification statements.
- Relation to existing slicing approaches.
- Formal proofs.

Summary

- Theory for diagnostics, suitable for *classical and probabilistic programs*:
  (1) **Error Localization** by error-witnessing slices (slice has error $\Rightarrow$ original has same error),
  (2) **Certificates** by verification-witnessing slices (slice verifies $\Rightarrow$ original verifies),
  (3) **Hints** by verification-preserving slices (tailoring program to specification).
- For *quantitative* intermediate verification language (IVL), and with specialized diagnostics.
- Implemented in Caesar tool; demonstrated on benchmarks.

Paper accepted at *ESOP 2026* (preprint online): *"Error Localization, Certificates, and Hints for Probabilistic Program Verification via Slicing"*.

- Upper bounds (e.g. `coassert`), specification statements.
- Relation to existing slicing approaches.
- Formal proofs.

Thu 10:45 - 11:10 @ POPL: *"Verifying Almost-Sure Termination for Randomized Distributed Algorithms"* by Enea et al., using Caesar!