Automated Hyperbug Finding

A. Correnson, T. Nießen, B. Finkbeiner, G. Weissenbacher
July 23, 2024

Part I: Motivation

Motivating Example

Let's look at a simple voting protocol for two candidates A and B...

$$
\begin{array}{ll}
\ell_0: & countA \leftarrow 0 \\
\ell_1: & countB \leftarrow 0 \\
\ell_2: & \textbf{loop} \\
\ell_3: & \quad \textbf{input } vote \in \{A, B\} \\
\ell_4: & \quad \textbf{if } vote = A \textbf{ then} \\
\ell_5: & \quad\quad countA \leftarrow countA + 1 \\
\ell_6: & \quad \textbf{if } vote = B \textbf{ then} \\
\ell_7: & \quad\quad countB \leftarrow countB + 1
\end{array}
$$

Motivating Example

… there could be a *tiny bug* in its implementation that plays in the favor of candidate B (whoops)

$$
\begin{array}{ll}
\ell_0: & countA \leftarrow 0 \\
\ell_1: & countB \leftarrow 0 \\
\ell_2: & \textbf{loop} \\
\ell_3: & \quad \textbf{input } vote \in \{A, B\} \\
\ell_4: & \quad \textbf{if } vote = A \textbf{ then} \\
\ell_5: & \quad\quad countA \leftarrow countA + 1 \\
\ell_6: & \quad \textbf{if } vote = B \textbf{ then} \\
\ell_7: & \quad\quad countB \leftarrow \color{red}{countA + 1}
\end{array}
$$

A subtle bug

Observations

A subtle bug

Observations

1. This bug is surprisingly difficult to catch with traditional bug-finding tools without giving a full formal specification of the voting protocol.

A subtle bug

Observations

1. This bug is surprisingly difficult to catch with traditional bug-finding tools without giving a full formal specification of the voting protocol.
2. Even without a precise specification, it is clear that this code cannot be correct because it does not treat candidates A and B equally.

A subtle bug

Observations

1. This bug is surprisingly difficult to catch with traditional bug-finding tools without giving a full formal specification of the voting protocol.
2. Even without a precise specification, it is clear that this code cannot be correct because it does not treat candidates A and B equally.

"candidates A and B should be treated equally" is an example of a hyperproperty
$\Longrightarrow$ it requires comparing several executions of the voting protocol

Specifying Hyperproperties in HyperLTL

To formally specify hyperproperties of software systems, we use the logic HyperLTL, an extension of LTL with trace quantification.

Specifying Hyperproperties in HyperLTL

To formally specify hyperproperties of software systems, we use the logic HyperLTL, an extension of LTL with trace quantification.

Trace quantification

$$\psi ::= \forall \tau . \psi \mid \exists \tau . \psi \mid \varphi$$

where φ is a temporal relation between traces

Specifying Hyperproperties in HyperLTL

To formally specify hyperproperties of software systems, we use the logic HyperLTL, an extension of LTL with trace quantification.

Trace quantification

$$\psi ::= \forall \tau . \psi \mid \exists \tau . \psi \mid \varphi$$

where φ is a temporal relation between traces

Temporal relations

$$\varphi ::= P \mid \varphi \land \varphi \mid \Box \varphi \mid \varphi \ \mathsf{U} \ \varphi \mid \ldots$$

where P is a predicate over program variables labeled by a trace variable τ

Specifying Asynchronous Hyperproperties in OHyperLTL

OHyperLTL extends HyperLTL with support for asynchronous reasoning by introducing explicit observation points

Specifying Asynchronous Hyperproperties in OHyperLTL

OHyperLTL extends HyperLTL with support for asynchronous reasoning by introducing explicit observation points

$$\psi ::= \forall \tau : o.\psi \mid \exists \tau : o.\psi \mid \varphi$$

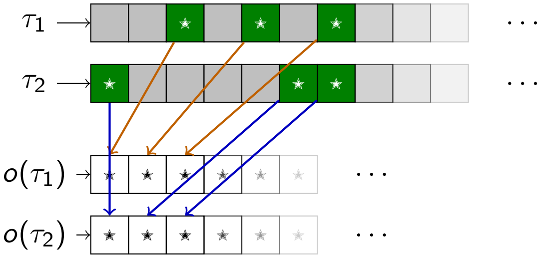
where o is a predicate specifying which states should be observed

Specifying Asynchronous Hyperproperties in OHyperLTL

OHyperLTL extends HyperLTL with support for asynchronous reasoning by introducing explicit observation points

$$\psi ::= \forall \tau : o.\psi \mid \exists \tau : o.\psi \mid \varphi$$

where o is a predicate specifying which states should be observed



OHyperLTL in Action

Going back to our example of the voting protocol, one way to formalize the intuition "candidates A and B should be treated equally" is by requiring for the protocol to be symmetric.

OHyperLTL in Action

Going back to our example of the voting protocol, one way to formalize the intuition "candidates A and B should be treated equally" is by requiring for the protocol to be symmetric.

$$\begin{aligned}\forall \tau_1 : \ell_3.\ \forall \tau_2 : \ell_3.\ &\Box(vote_{\tau_1} = A \leftrightarrow vote_{\tau_2} = B) \rightarrow \\ &\Box(countA_{\tau_1} = countB_{\tau_2} \land countB_{\tau_1} = countA_{\tau_2})\end{aligned}$$

OHyperLTL in Action

Going back to our example of the voting protocol, one way to formalize the intuition "candidates A and B should be treated equally" is by requiring for the protocol to be symmetric.

$$\forall \tau_1 : \ell_3 .\ \forall \tau_2 : \ell_3 .\ \Box(vote_{\tau_1} = A \leftrightarrow vote_{\tau_2} = B) \rightarrow \\ \Box(countA_{\tau_1} = countB_{\tau_2} \wedge countB_{\tau_1} = countA_{\tau_2})$$

Another, simpler, way is to require that candidates have equal opportunities

OHyperLTL in Action

Going back to our example of the voting protocol, one way to formalize the intuition "candidates A and B should be treated equally" is by requiring for the protocol to be symmetric.

$$\begin{aligned}\forall \tau_1 : \ell_3.\ \forall \tau_2 : \ell_3.\ \Box(vote_{\tau_1} = A \leftrightarrow vote_{\tau_2} = B) \rightarrow \\ \Box(countA_{\tau_1} = countB_{\tau_2} \wedge countB_{\tau_1} = countA_{\tau_2})\end{aligned}$$

Another, simpler, way is to require that candidates have equal opportunities

$$\forall \tau_1 : \ell_3.\ \exists \tau_2 : \ell_3.\ \Box(countA_{\tau_1} = countB_{\tau_2} \wedge countA_{\tau_2} = countB_{\tau_1})$$

OHyperLTL in Action

Going back to our example of the voting protocol, one way to formalize the intuition "candidates A and B should be treated equally" is by requiring for the protocol to be symmetric.

$$\forall \tau_1 : \ell_3.\ \forall \tau_2 : \ell_3.\ \square(vote_{\tau_1} = A \leftrightarrow vote_{\tau_2} = B) \rightarrow \\ \square(countA_{\tau_1} = countB_{\tau_2} \wedge countB_{\tau_1} = countA_{\tau_2})$$

Another, simpler, way is to require that candidates have equal opportunities

$$\forall \tau_1 : \ell_3.\ \exists \tau_2 : \ell_3.\ \square(countA_{\tau_1} = countB_{\tau_2} \wedge countA_{\tau_2} = countB_{\tau_1})$$

Importantly, the alternation of universal and existential quantification enables a concise specification that does not refer to the inputs of the protocol

Summary of observations made so far

Towards automated bug-hunting for OHyperLTL?

Summary of observations made so far

1. Some subtle bugs can only be detected by checking programs against hyperproperties

Summary of observations made so far

1. Some subtle bugs can only be detected by checking programs against hyperproperties
2. Alternation of $\forall$ and $\exists$ trace quantifiers is a convenient/necessary feature for concise specification of relevant hyperproperties

Towards automated bug-hunting for OHyperLTL?

Summary of observations made so far

1. Some subtle bugs can only be detected by checking programs against hyperproperties
2. Alternation of $\forall$ and $\exists$ trace quantifiers is a convenient/necessary feature for concise specification of relevant hyperproperties
3. In the context of software, hyperproperties we wish to specify are asynchronous

Towards automated bug-hunting for OHyperLTL?

Summary of observations made so far

1. Some subtle bugs can only be detected by checking programs against hyperproperties
2. Alternation of $\forall$ and $\exists$ trace quantifiers is a convenient/necessary feature for concise specification of relevant hyperproperties
3. In the context of software, hyperproperties we wish to specify are asynchronous

Our goal

$\rightarrow$ a fully automated bug-hunting technique for $\forall^*\exists^*$ asynchronous hyperproperties expressed in OHyperLTL

Verification is extremely difficult

→ requires finding a proof that, for every first trace, there exists a second trace that satisfies the specified relation

Challenges

Verification is extremely difficult

→ requires finding a proof that, for every first trace, there exists a second trace that satisfies the specified relation

Refutation is not (much) simpler

→ requires finding a trace and a proof that, for this trace, no second trace exists that satisfies the specified relation

State of the art

Many existing approaches even for $\forall\exists$ hyperproperties, but...

- Game-based verification
  $\rightarrow$ incomplete and does not produce counterexamples

State of the art

Many existing approaches even for $\forall\exists$ hyperproperties, but...

- Game-based verification
  $\rightarrow$ incomplete and does not produce counterexamples
- Hoare-style relational verification
  $\rightarrow$ requires expert guidance (loop invariants, predicate abstractions, ...)

State of the art

Many existing approaches even for $\forall\exists$ hyperproperties, but...

- Game-based verification
  $\rightarrow$ incomplete and does not produce counterexamples
- Hoare-style relational verification
  $\rightarrow$ requires expert guidance (loop invariants, predicate abstractions, ...)
- Automata-based model-checking and QBF-based bounded model-checking
  $\rightarrow$ limited to the analysis of finite-state systems

State of the art

Many existing approaches even for $\forall\exists$ hyperproperties, but...

- Game-based verification
  $\rightarrow$ incomplete and does not produce counterexamples
- Hoare-style relational verification
  $\rightarrow$ requires expert guidance (loop invariants, predicate abstractions, ...)
- Automata-based model-checking and QBF-based bounded model-checking
  $\rightarrow$ limited to the analysis of finite-state systems

State of the art

Many existing approaches even for $\forall\exists$ hyperproperties, but...

- Game-based verification
  $\rightarrow$ incomplete and does not produce counterexamples
- Hoare-style relational verification
  $\rightarrow$ requires expert guidance (loop invariants, predicate abstractions, ...)
- Automata-based model-checking and QBF-based bounded model-checking
  $\rightarrow$ limited to the analysis of finite-state systems

$\rightarrow$ no existing approach can fully automatically find counterexamples to $\forall\exists$ hyperproperties in asynchronous, infinite-state systems

Part II: Symbolic Execution for Asynchronous Hyperproperties

Taking a step back: Symbolic execution for traditional bug-finding

Symbolic execution explores all behavior of a program
by computing a symbolic encoding of the program's paths.

Taking a step back: Symbolic execution for traditional bug-finding

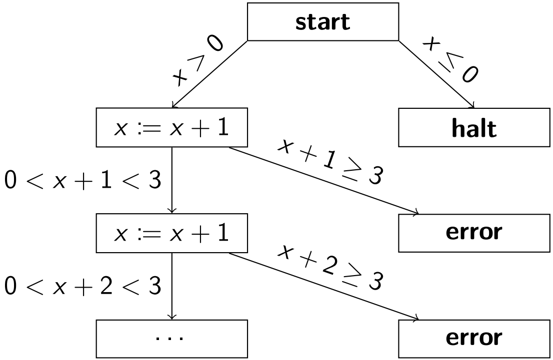
Symbolic execution explores all behavior of a program
by computing a symbolic encoding of the program's paths.

while $x > 0$
  $x := x + 1$
  assert $x < 3$

Taking a step back: Symbolic execution for traditional bug-finding

Symbolic execution explores all behavior of a program
by computing a symbolic encoding of the program's paths.

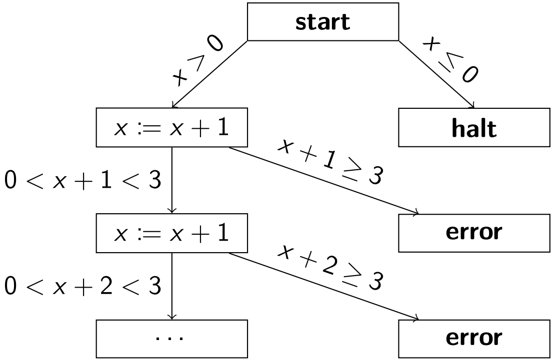
while $x > 0$
  $x := x + 1$
  assert $x < 3$



Taking a step back: Symbolic execution for traditional bug-finding

Symbolic execution explores all behavior of a program
by computing a symbolic encoding of the program's paths.

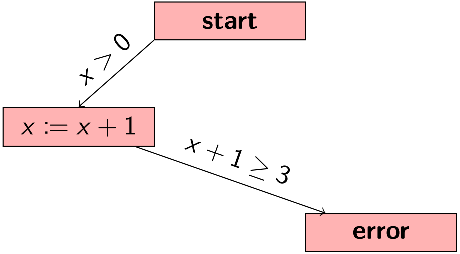
- → logical representation of program paths
- → reduces bug finding to SMT solving



Taking a step back: Symbolic execution for traditional bug-finding

Symbolic execution explores all behavior of a program
by computing a symbolic encoding of the program's paths.

$0 < x \land x + 1 \geq 3$ is sat
$\implies$ bug found!



Symbolic Execution for OHyperLTL$_{\text{safe}}$

Given: OHyperLTL$_{\text{safe}}$ hyperproperty $\forall\tau_1.\exists\tau_2.\Box\varphi$

Idea: use two symbolic execution engines to find a model for $\exists\tau_1.\forall\tau_2.\neg\Box\varphi$

Symbolic Execution for $\text{OHyperLTL}_{\text{safe}}$

Given: $\text{OHyperLTL}_{\text{safe}}$ hyperproperty $\forall \tau_1.\exists \tau_2.\Box \varphi$

Idea: use two symbolic execution engines to find a model for $\exists \tau_1.\forall \tau_2.\neg \Box \varphi$

- one symbolic execution engine searches for candidate symbolic paths π_1
  (for a bounded number of observation points)

Symbolic Execution for $\text{OHyperLTL}_{\text{safe}}$

Given: $\text{OHyperLTL}_{\text{safe}}$ hyperproperty $\forall \tau_1 . \exists \tau_2 . \Box \varphi$

Idea: use two symbolic execution engines to find a model for $\exists \tau_1 . \forall \tau_2 . \neg \Box \varphi$

- one symbolic execution engine searches for candidate symbolic paths π_1 (for a bounded number of observation points)
- a second symbolic execution engine tries to refute that π_1 is a counterexample by searching for a path π_2 such that $\Box \varphi$ holds for π_1, π_2

Symbolic Execution for $\text{OHyperLTL}_{\text{safe}}$

Given: $\text{OHyperLTL}_{\text{safe}}$ hyperproperty $\forall\tau_1.\exists\tau_2.\square\varphi$

Idea: use two symbolic execution engines to find a model for $\exists\tau_1.\forall\tau_2.\neg\square\varphi$

- one symbolic execution engine searches for candidate symbolic paths π_1 (for a bounded number of observation points)
- a second symbolic execution engine tries to refute that π_1 is a counterexample by searching for a path π_2 such that $\square\varphi$ holds for π_1, π_2

$\rightarrow$ when this refutation fails, we have found a hyperbug!

Symbolic Execution for $\text{OHyperLTL}_{\text{safe}}$

Given: $\text{OHyperLTL}_{\text{safe}}$ hyperproperty $\forall\tau_1.\exists\tau_2.\Box\varphi$

Idea: use two symbolic execution engines to find a model for $\exists\tau_1.\forall\tau_2.\neg\Box\varphi$

- one symbolic execution engine searches for candidate symbolic paths π_1 (for a bounded number of observation points)
- a second symbolic execution engine tries to refute that π_1 is a counterexample by searching for a path π_2 such that $\Box\varphi$ holds for π_1, π_2

$\rightarrow$ when this refutation fails, we have found a hyperbug!

Symbolic Execution for OHyperLTL$_{\text{safe}}$

Given: OHyperLTL$_{\text{safe}}$ hyperproperty $\forall \tau_1 . \exists \tau_2 . \Box \varphi$

Idea: use two symbolic execution engines to find a model for $\exists \tau_1 . \forall \tau_2 . \neg \Box \varphi$

- one symbolic execution engine searches for candidate symbolic paths π_1 (for a bounded number of observation points)
- a second symbolic execution engine tries to refute that π_1 is a counterexample by searching for a path π_2 such that $\Box \varphi$ holds for π_1, π_2

$\rightarrow$ when this refutation fails, we have found a hyperbug!

Can be generalized to $\forall^* \exists^*$ hyperproperties through product constructions

The good news

Theorem (Soundnesss)

Symbolic Execution for $OHyperLTL_{\text{safe}}$ *is a sound hyperbug finding method.*

The good news

Theorem (Soundnesss)

Symbolic Execution for $OHyperLTL_{\text{safe}}$ is a sound hyperbug finding method.

The bad news

Lemma (Undecidability)

Refuting asynchronous $\forall\exists$ hyperproperties of infinite-state systems is undecidable.
(reduction from the halting problem)

Symbolic Execution for OHyperLTL$_{\mathsf{safe}}$

The good news

Theorem (Soundnesss)

Symbolic Execution for $OHyperLTL_{\mathsf{safe}}$ is a sound hyperbug finding method.

The bad news

Lemma (Undecidability)

Refuting asynchronous $\forall\exists$ hyperproperties of infinite-state systems is undecidable.
(reduction from the halting problem)

Corollary (Incompleteness)

Symbolic Execution for $OHyperLTL_{\mathsf{safe}}$ is necessarily incomplete.

What is the problem?

Intuition: using symbolic execution to find a model for $\exists \tau_1 . \forall \tau_2 . \neg \Box \varphi$ requires to enumerate all possible symbolic paths of a given observation length

What is the problem?

Intuition: using symbolic execution to find a model for $\exists \tau_1 . \forall \tau_2 . \neg \Box \varphi$ requires to enumerate all possible symbolic paths of a given observation length

The set of symbolic paths of observational length k is not necessarily finite!
$\implies$ we cannot always enumerate symbolic paths of observation length k

What is the problem?

Intuition: using symbolic execution to find a model for $\exists \tau_1 . \forall \tau_2 . \neg \Box \varphi$ requires to enumerate all possible symbolic paths of a given observation length

The set of symbolic paths of observational length k is not necessarily finite!
$\Longrightarrow$ we cannot always enumerate symbolic paths of observation length k

ℓ_0: **loop**
ℓ_1: **while**$(\dots)$ $\{\dots\}$
ℓ_2: **observe**

Symbolic paths of observational lengths k are of the form $(\ell_0(\ell_1)^\star \ell_2)^k$ (i.e., there are infinitely many of them)

Relative completeness for observable programs

Definition (Observable Programs)

A program is observable if, in any symbolic state, there is some n such that, after n steps, the program either produces an observed system state or terminates.

Relative completeness for observable programs

Definition (Observable Programs)

A program is observable if, in any symbolic state, there is some n such that, after n steps, the program either produces an observed system state or terminates.

Theorem

Relative Completeness Symbolic Execution for $\mathrm{OHyperLTL}_{\mathrm{safe}}$ *is a complete hyperbug finding method for observable programs (assuming symbolic paths can be expressed in a decidable first-order theory).*

Part III: Evaluation

Experimental results - ORHLE benchmarks

Class	Type	Program	FO	Bug found	# Combinations	Runtime
$\forall\exists$	Other	`draw-once`	✓	✓	1	0.001 s
$\forall\exists$	Refinement	`simple-nonrefinement`	✓	✓	1	0.001 s
$\forall\exists$	Other	`do-nothing`	✓	✓	1	0.001 s
$\forall\forall\exists$	Generalized non-interference	`nondet-leak2`	✓	✓	2	0.001 s
$\forall\forall\exists$	Generalized non-interference	`simple-leak`	✓	✓	1	0.001 s
$\forall\forall\exists$	Generalized non-interference	`smith1`	✓	✓	2	0.003 s
$\forall\forall\exists$	Generalized non-interference	`nondet-leak`	✓	✓	2	0.003 s
$\forall\forall\exists$	Delimited release	`parity-no-dr`	✓	✓	2	0.003 s
$\forall\forall\exists$	Delimited release	`wallet-no-dr`	✓	✓	2	0.003 s
$\forall\exists$	Refinement	`conditional-nonrefinement`	✓	✓	4	0.006 s
$\forall\exists$	Refinement	`add3-shuffled`	✓	✓	6	0.009 s
$\forall\forall\forall\exists$	Delimited release	`conditional-no-dr`	✓	✓	8	0.013 s
$\forall\forall\exists$	Delimited release	`median-no-dr`	✓	✓	4	0.016 s
$\forall\forall\forall\exists$	Generalized non-interference	`conditional-leak`	✓	✓	48	0.074 s
$\forall\exists$	Refinement	`loop-nonrefinement`	✗	✗	N/A	∞

Experimental results - our own benchmarks

Class	Program	Bug found	# Observations	# Combinations	Runtime
$\forall\exists$	`even_odd`	✓	1	1	0.001 s
$\forall\exists$	`factor2`	✓	2	2	0.001 s
$\forall\exists$	`for_loop_simple`	✓	1	2	0.010 s
$\forall\exists$	`linear_equation`	✓	22	22	0.023 s
$\forall\exists$	`monotonic_increase`	✓	7	7	0.029 s
$\forall\exists$	`escalating`	✓	7	747	0.103 s
$\forall\forall\exists$	`secret_pin_leak`	✓	8	11	0.103 s
$\forall\exists$	`escalating_2`	✓	7	1707	0.190 s
$\forall\forall$	`obs_determinism`	✓	4	86	0.408 s
$\forall\exists$	`no_primes_above_31397`	✓	1	201	1.203 s
$\forall\forall\exists$	`secret_pin_leak_2`	✓	3	248	1.972 s
$\forall\exists$	`exponential_branching_1`	✓	1	1024	2.216 s
$\forall\exists$	`exponential_branching_2`	✓	1	2048	4.040 s