Dynamic Logic for Practical Program Verification

Set 4

Wolfgang Ahrendt

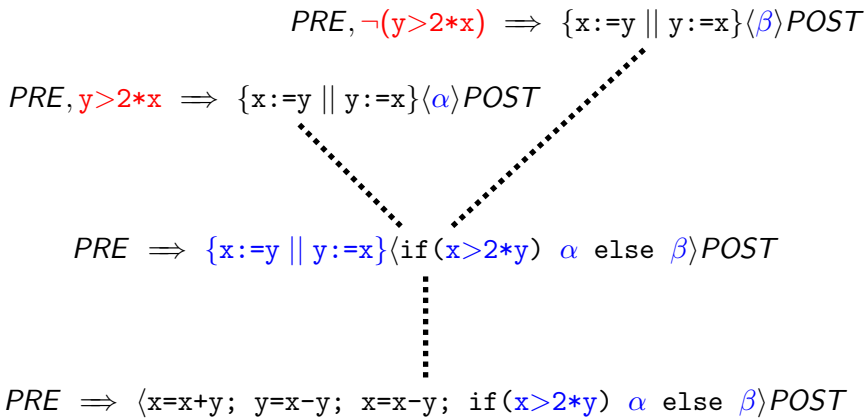
Chalmers University of Technology, Gothenburg, Sweden

VTSA Summer School, Luxembourg, 2024

Part I

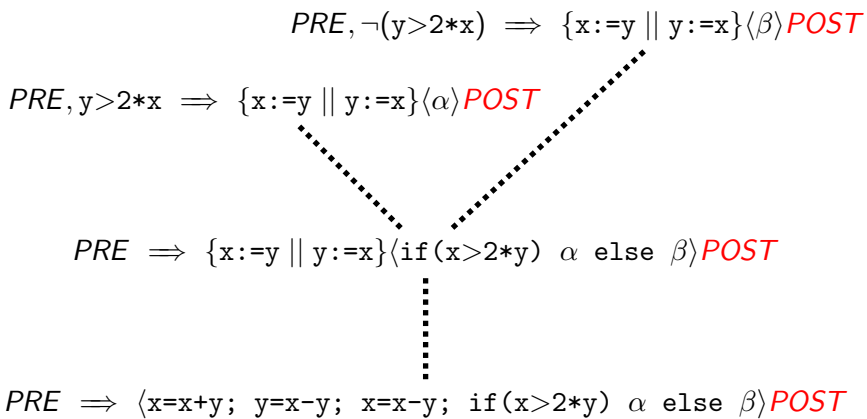
Verification Based Test Case Generation

Recall Generation of Path Conditions

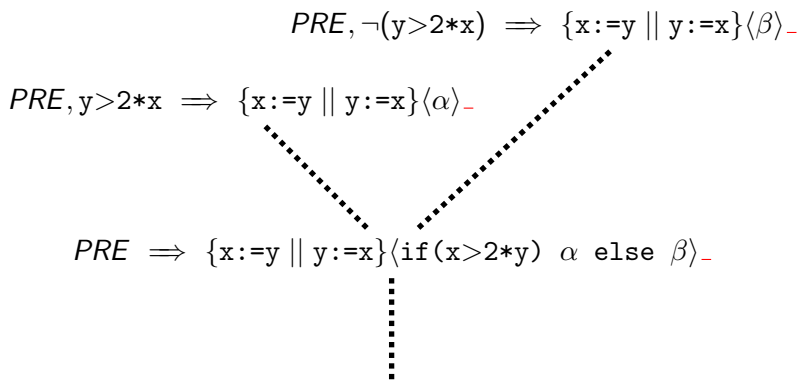


y>2*x and ¬(y>2*x) are path conditions on prestate

Postcondition not used for Generating Test Inputs



Postcondition not used for Generating Test Inputs



From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint
   - Generate data satisfying the constraint
     (Model Generation, SMT solving)

From (partial) Proofs to Test Cases

1. Extract 'Execution Tree'
   (stripped down proof tree)

2. For each leaf, generate at least one test
   - Isolate input constraint
   - Generate data satisfying the constraint
     (Model Generation, SMT solving)
   - Construct test input(s) in JUnit format

Achieved Coverage Criteria

- Structural Coverage Criteria
  - Statement Coverage
  - Branch Coverage
- Logical Coverage Criteria
  - Top level Boolean Decision Coverage (DC)
  - Atomic Boolean Condition Coverage (CC)
  - Multiple Boolean Condition Coverage (MCC)
    (all possible combinations)
  - Modified Condition/Decision Coverage (MCDC)
    (required in Avionics certification standards)

Test Generation Case Study

- applied to implementations of 40 methods in Real-Time Java API
- KeYTestGen produced a failing test case,
  i.e., detected a bug in Real-Time Java library code

References

- Christian Engel, Reiner Hähnle
  Generating Unit Tests from Formal Proofs
  Testing and Proofs 2007
  Springer LNCS 4454
- Wolfgang Ahrendt, Wojciech Mostowski, Gabriele Paganelli
  Real-time Java API specifications for high coverage test generation
  JTRES '12
  ACM

Part II

Combining Static and Runtime Verification

Project on Combined Static and Runtime Verification

STAtic and Runtime Verification of Object-ORiented SW

Project on Combined Static and Runtime Verification

STAtic and Runtime Verification of Object-ORiented SW

STARVOORS

Runtime Verification vs. Static Verification

- Runtime verification
  - Full automation

but

  - Cannot judge future runs
  - Runtime overhead

Runtime Verification vs. Static Verification

- Runtime verification
  - Full automation

  but
  - Cannot judge future runs
  - Runtime overhead
- Static verification
  - Reason about properties of all possible runs
  - No runtime overhead

  but
  - Hard to achieve full automation (e.g. invariants)
  - May loose aspects of concrete system

Approach

StaRVOOrS approach:

- Combine static and runtime verification
  - Combine data centric and control centric properties
  - Unified specification for both
- Use (partial) static verification results for partial evaluation of properties
- Runtime verification of resulting properties

Approach

StaRVOOrS approach:

- Combine static and runtime verification
  - Combine data centric and control centric properties
  - Unified specification for both
- Use (partial) static verification results for partial evaluation of properties
- Runtime verification of resulting properties
- Language target: Java

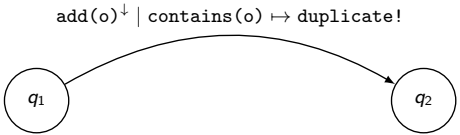
Language and Underlying Tools

- A specification language for Data and Control Properties: *ppDATE*
- Combining static and runtime verification:
  - KeY
    - Deductive verification, i.e., using theorem proving
    - Properties are written in JML
  - LARVA
    - Extended Automata for Runtime Verification
    - Properties are written in *DATE*

Language and Underlying Tools

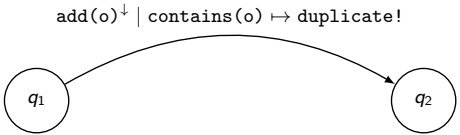
- A specification language for Data and Control Properties: *ppDATE* (extension of *DATE* w. *p*re/*p*ost-conditions)
- Combining static and runtime verification:
  - KeY
    - Deductive verification, i.e., using theorem proving
    - Properties are written in JML
  - LARVA
    - Extended Automata for Runtime Verification
    - Properties are written in *DATE*

ppDATE Example: Scenario including a Hashtable



q_1 'contains': $\{\texttt{size} < \texttt{capacity}\}\,\texttt{add(o)}\,\{\exists\, i.\texttt{arr}[i] = \texttt{o}\}$

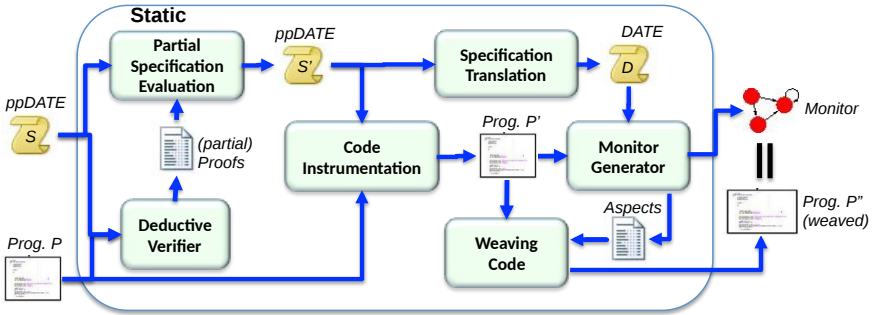
ppDATE Example: Scenario including a Hashtable



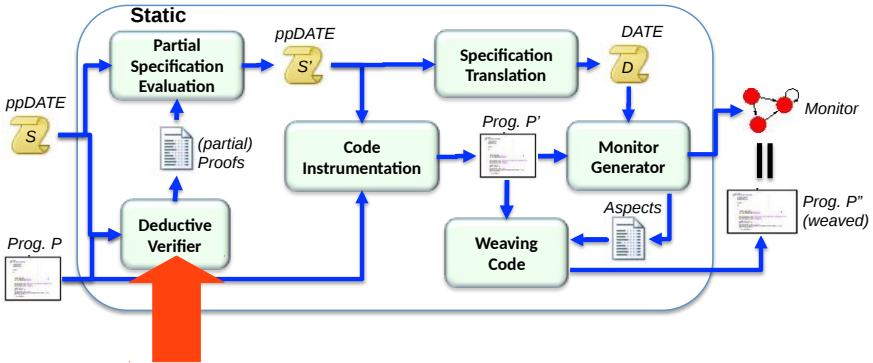
q_1 'contains': $\{\texttt{size} < \texttt{capacity}\}\ \texttt{add(o)}\ \{\exists\ i.\texttt{arr}[i] = \texttt{o}\}$

- Hoare triples are described using JML-like notation

High-level description of the framework



High-level description of the framework



Static Verification Task

- Implementation of `add()`:

```
public void add(Object u) {
   int hashcode = u.hashCode();
   int key = hashcode % capacity;
   if arr[key] == null {
      arr[key] = u;
      size++;
   }
   else
      while ... \\ store at next free slot
}
```

Static Verification Task

- Implementation of `add()`:

```
public void add(Object u) {
   int hashcode = u.hashCode();
   int key = hashcode % capacity;
   if arr[key] == null {
      arr[key] = u;
      size++;
   }
   else
      while ... \\ store at next free slot
}
```

- We attempt to statically prove:

$$\{\texttt{size} < \texttt{capacity}\}\ \texttt{add(o)}\ \{\exists\, i.\ \texttt{arr}[i] = \texttt{o}\}$$

Deductive Verifier

- KeY tries to prove:

$$\{\texttt{size} < \texttt{capacity}\}\ \texttt{add(o)}\ \{\exists i.\ \texttt{arr}[i] = \texttt{o}\}$$

- KeY cannot fully prove (automatically)
- proof branch

$$\ldots, \texttt{arr[hashcode\%capacity]} = \texttt{null} \implies \ldots$$

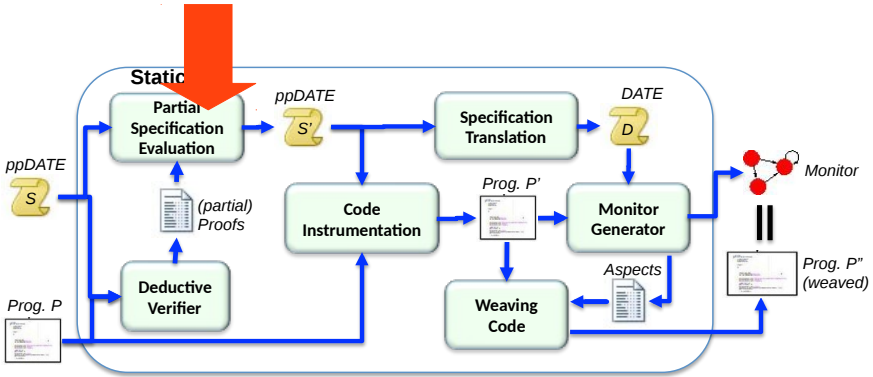
closed (automatically)

- proof branch

$$\ldots, \neg\, \texttt{arr[hashcode\%capacity]} = \texttt{null} \implies \ldots$$

not closed (automatically)

High-level description of the framework



Partial Specification Evaluation

- partial proof gives path condition

$$\neg\ \texttt{arr[hashcode\%capacity]} = \texttt{null}$$

to be added to pre-condition:

$$\{\mathit{pre} \wedge \neg\ \texttt{arr[hashcode\%capacity]} = \texttt{null}\}\ \texttt{add(o)}\ \{\mathit{post}\}$$

Partial Specification Evaluation

- partial proof gives path condition

$$\neg\ \texttt{arr[hashcode\%capacity]} = \texttt{null}$$

to be added to pre-condition:

$$\{\mathit{pre} \wedge \neg\ \texttt{arr[hashcode\%capacity]} = \texttt{null}\}\ \texttt{add(o)}\ \{\mathit{post}\}$$

- Monitor *this* specification instead

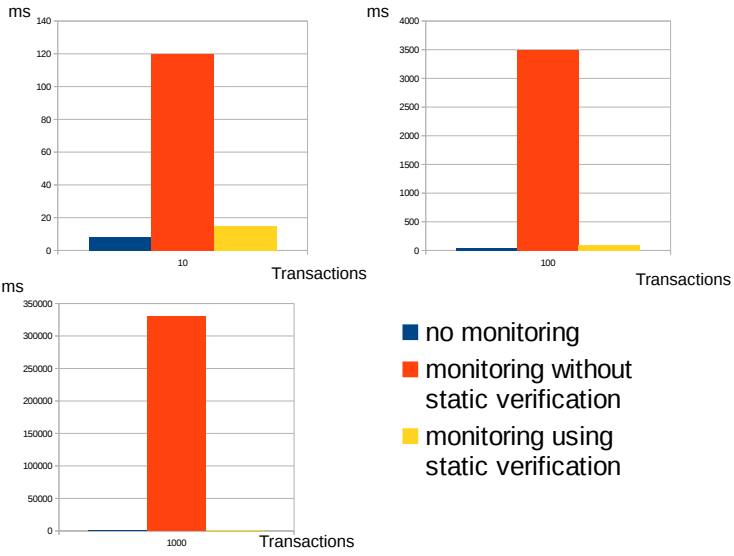
Mondex Case Study

- Electronic purse application
- Financial transaction move funds between accounts
- Multi-step message exchange protocol
  - *ppDATE* specification has 10 states and 25 transitions
  - 26 different Hoare-triples

Mondex Case Study

Transactions	*no* monitoring	monitoring without static verif.	monitoring using static verif.
10	8 ms	120 ms	15 ms
100	50 ms	3,500 ms	90 ms
1000	250 ms	330,000 ms	375 ms

Mondex Case Study



References

- W. Ahrendt, M. Chimento, G. Pace, G. Schneider
  Verifying Data- and Control-oriented Properties Combining Static and Runtime Verification: Theory and Tools
  Formal Methods in System Design, 2017

Part III

Hybrid Systems Verification

Dynamic Logic for Hybrid Systems

Hybrid System Models

Mathematical models for digital/physical systems with

- discrete dynamics (system+control decisions, environment choices)
- continuous dynamics (differential equations for system evolution)

DL for Hybrid System

- Differential dynamic logic (dL): modelling and specification of hybrid systems
- KeYmaera X: theorem prover for dL
- dL and KeYmaera X invented by André Platzer
- Book: A. Platzer, *Logical Foundations of Cyber-Physical Systems*, Springer

Hybrid Programs (HP)

Statement	Effect
$x := e$	discrete assignment of e to x
$x := *$	discrete nondeterministic assignment of arbitrary real number to x
$?Q$	skip if Q is true, fail otherwise
$x' = f(x)$ & Q	continuous evolution along differential equation system $x' = f(x)$, for nondeterministic duration (as long as Q is true)
$\alpha; \beta$	sequential composition
$\alpha \cup \beta$	nondeterministic choice, follow either α or β
α^*	nondeterministic repetition, repeat α n times for any $n \in \mathbb{N}_0$

Hybrid Programs (HP)

Statement	Effect
$x := e$	discrete assignment of e to x
$x := *$	discrete nondeterministic assignment of arbitrary real number to x
$?Q$	skip if Q is true, fail otherwise
$x' = f(x)$ & Q	continuous evolution along differential equation system $x' = f(x)$, for nondeterministic duration (as long as Q is true)
$\alpha; \beta$	sequential composition
$\alpha \cup \beta$	nondeterministic choice, follow either α or β
α^*	nondeterministic repetition, repeat α n times for any $n \in \mathbb{N}_0$

Note: $x := *; ?Q$ allows declarative programming: "choose x such that Q"

Differential Dynamic Logic (dL)

Differential dynamic logic formulas (dL)

first-order logic over $\mathbb{R}$ plus modal operators $[\alpha]$ and $\langle\alpha\rangle$

- $[\alpha]\, P$ *all* (non-failing) executions of HP α end in a state satisfying P
- $\langle\alpha\rangle\, P$ *some* (non-failing) execution of HP α ends in a state satisfying P

Closed Loop System with Safety Controller

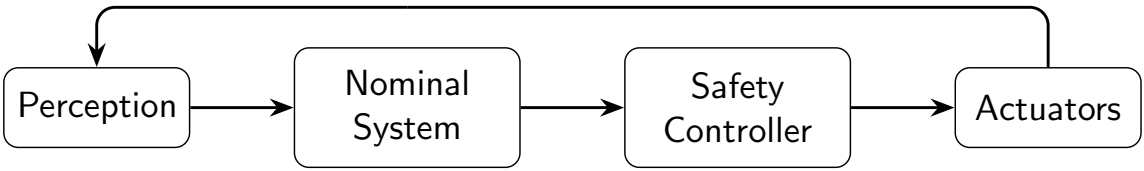


Figure: Closed loop.

Closed Loop System with Safety Controller

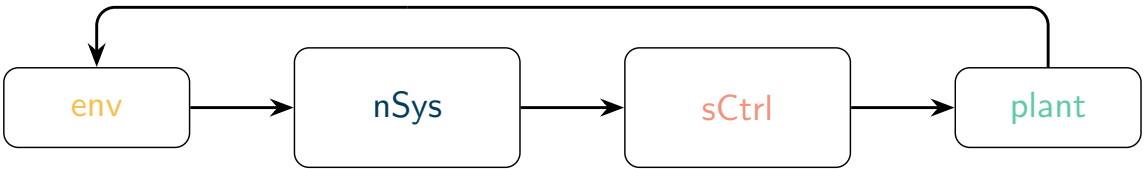


Figure: Closed loop.

$$(env;\ nSys;\ sCtrl;\ plant)^*$$

Closed Loop System with Safety Controller

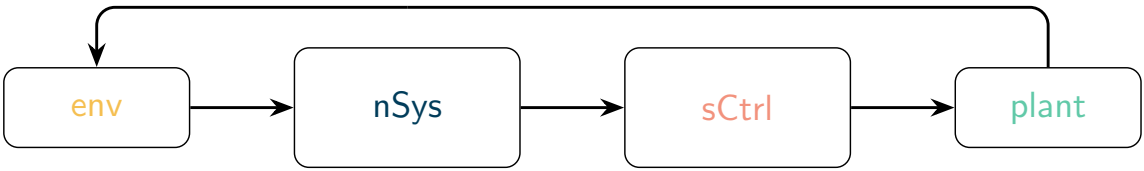


Figure: Closed loop.

$$init \rightarrow [(env;\ nSys;\ sCtrl;\ plant)^*]\, guarantee$$

Closed Loop System with Safety Controller

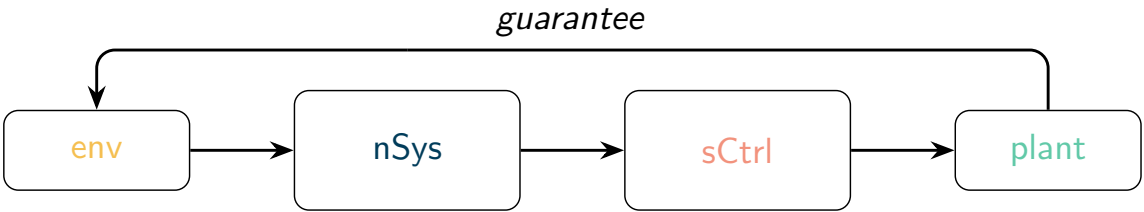


Figure: Closed loop.

$$init \rightarrow [(env;\ nSys;\ sCtrl;\ plant)^*]\, guarantee$$

Proving Control Loops

Loop invariant proof rule

To prove $init \rightarrow [(env;\ nSys;\ sCtrl;\ plant)^*]\ guarantee$, use loop invariant Inv:

1. $init \rightarrow Inv$
2. $Inv \rightarrow [\,env;\ nSys;\ sCtrl;\ plant\,]\ Inv$
3. $Inv \rightarrow guarantee$

Example: In-Lane Automated Driving System

$$init \rightarrow [(env;\ nSys;\ sCtrl;\ plant)^*]\, guarantee \tag{1}$$

$$env \triangleq x_c := *;\ ?\left(x_c - x \geq \frac{v^2}{2a^{min}}\right) \quad (\text{“environment assumptions”}) \tag{2}$$

$$nSys \triangleq a := *; \ldots\ldots \quad (\text{“potentially unsafe computation”}) \tag{3}$$

$$sCtrl \triangleq \text{if } \ldots(\text{“a is unsafe”})\ldots \text{ then } a := \ldots(\text{“safe computation”})\ldots \text{ fi} \tag{4}$$

$$plant \triangleq \tau := 0;\ x' = v, v' = a, \ldots\ldots\ \&\ \tau \leq T \tag{5}$$

Example: In-Lane Automated Driving System

Collaboration of Chalmers and Zenseact

Zenseact: software for autonomous driving and driver-assistance

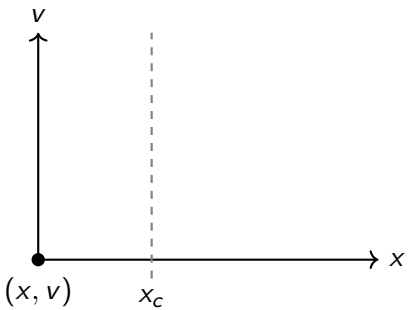
Case study:

- dL modeling and verification for safety argument of in-lane AD feature
- Formalise safety requirements as 'assume-guarantee' in dL
- Use formal analysis for safety requirement refinement
- Design and verify different refinements of the model
- Use theorem proving to compare verified models

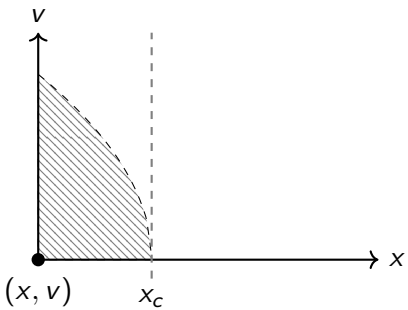
Reference

- Yuvaraj Selvaraj, Wolfgang Ahrendt, Martin Fabian
  Formal Development of Safe Automated Driving Using Differential Dynamic Logic
  IEEE Transactions on Intelligent Vehicles, 2023

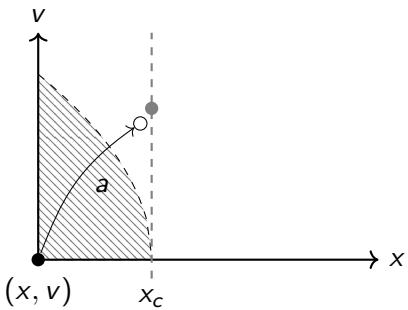
Potential Modelling Mistake (A): Exploiting Controller



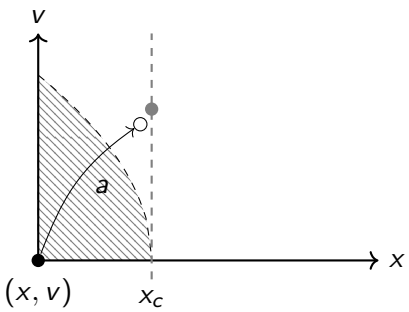
Potential Modelling Mistake (A): Exploiting Controller

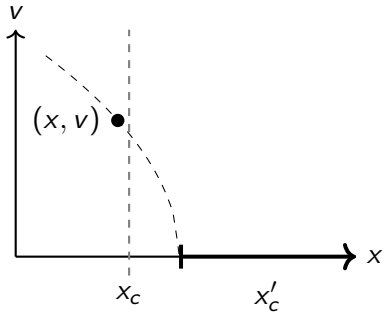


Potential Modelling Mistake (A): Exploiting Controller

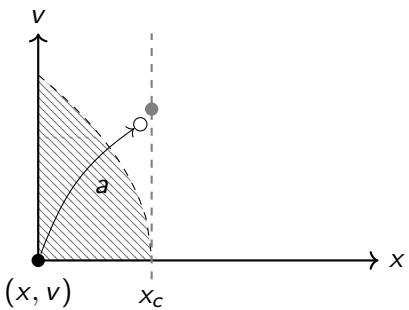


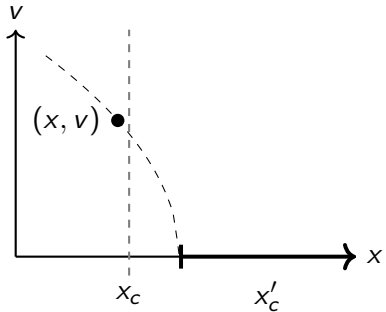
Potential Modelling Mistake (A): Exploiting Controller





Potential Modelling Mistake (A): Exploiting Controller

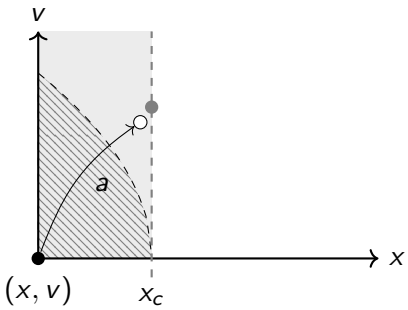


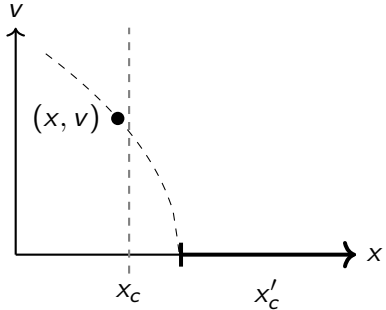


Exploiting controller

Controller forces environment to be too friendly.

Potential Modelling Mistake (A): Exploiting Controller





Exploiting controller

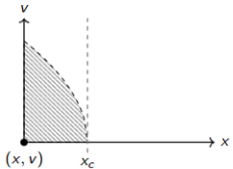
Controller forces environment to be too friendly.
Can be verified with invariant $Inv \equiv x \leq x_c$

Exploiting Controller

Unintended modelling behaviour: unsafe controller makes environment move obstacle away

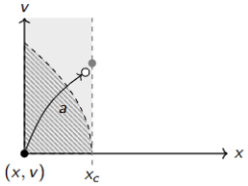
Exploiting Controller

Unintended modelling behaviour: unsafe controller makes environment move obstacle away



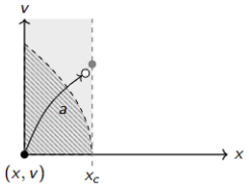
Exploiting Controller

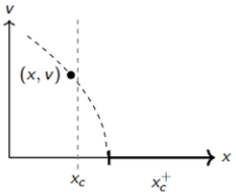
Unintended modelling behaviour: unsafe controller makes environment move obstacle away



Exploiting Controller

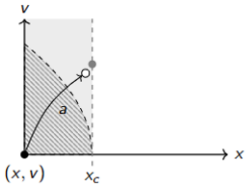
Unintended modelling behaviour: unsafe controller makes environment move obstacle away

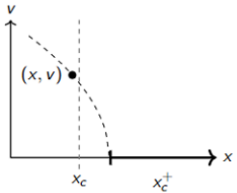




Exploiting Controller

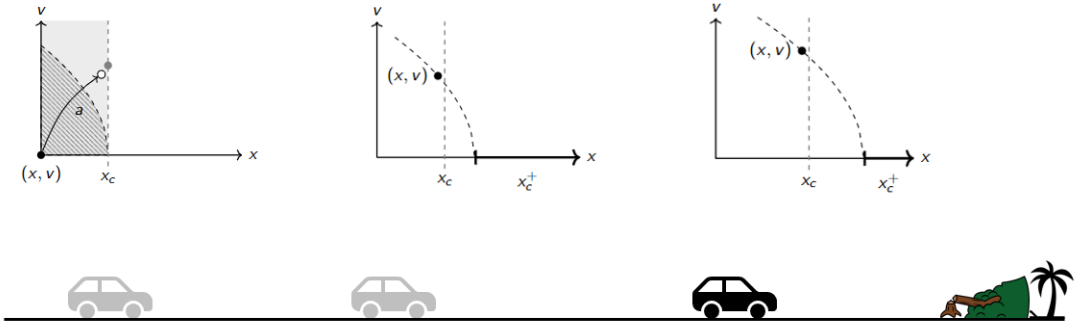
Unintended modelling behaviour: unsafe controller makes environment move obstacle away



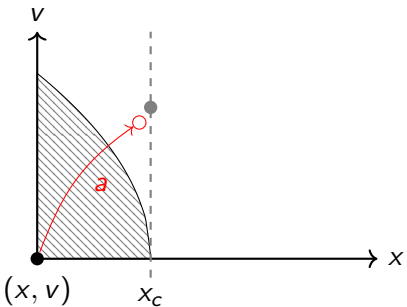


Exploiting Controller

Unintended modelling behaviour: unsafe controller makes environment move obstacle away



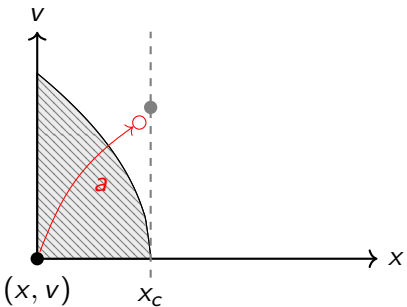
Solution



Idea

Strengthen invariant such that it allows the environment to change nothing

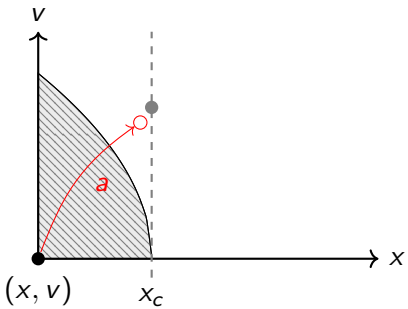
Solution



Idea

Strengthen invariant such that it allows the environment to change nothing

Solution



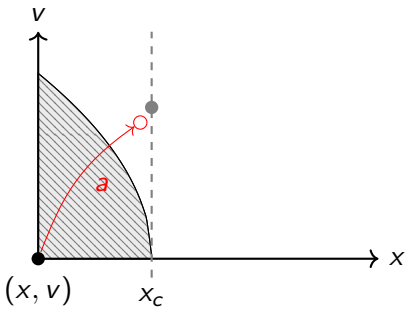
Idea

Strengthen invariant such that it allows the environment to change nothing

Proof Obligation (to be verified in KeYmaera)

$$\forall s.\, \forall e.\, \forall e_0.\, \Big(Inv(s, e) \wedge e = e_0 \rightarrow \langle env \rangle\, (e = e_0) \Big)$$

Solution



Idea

Strengthen invariant such that it allows the environment to change nothing

Consequence

Exploiting Controller will not verify, because it now breaks the invariant

Proof Obligation (to be verified in KeYmaera)

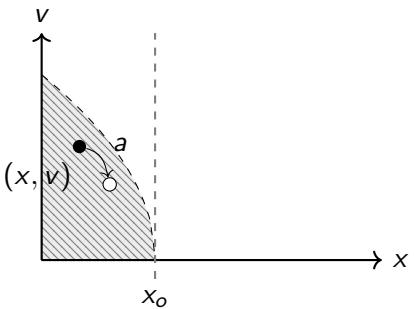
$$\forall s.\, \forall e.\, \forall e_0.\, \Big(Inv(s, e) \wedge e = e_0 \rightarrow \langle env \rangle\, (e = e_0) \Big)$$

Potential Modelling Mistake (B): Unchallenged Controller

Recall control loop: $(env; nSys; sCtrl; plant)^*$

Potential Modelling Mistake (B): Unchallenged Controller

Recall control loop: $(env; nSys; sCtrl; plant)^*$

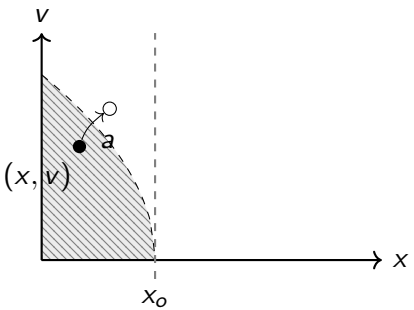


Consider the hybrid program without *sCtrl*:

$(env; nSys; plant)^*$.

Potential Modelling Mistake (B): Unchallenged Controller

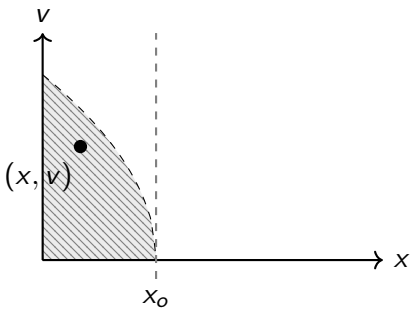
Recall control loop: $(env;\ nSys;\ sCtrl;\ plant)^*$



If all behaviours of $(env;\ nSys;\ plant)$ remain in the invariant states,
then *sCtrl* is vacuously verified.

Potential Modelling Mistake (B): Unchallenged Controller

Recall control loop: $(env; nSys; sCtrl; plant)^*$



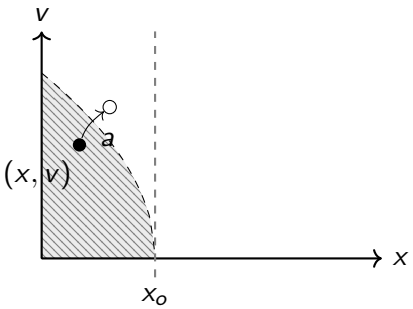
(Or, put equivalently:)
sCtrl is properly verified if there is at least one behaviour of (*env*; *nSys*; *plant*) that violates the invariant.

Solution

Recall control loop: $(env; nSys; sCtrl; plant)^*$

Solution

Recall control loop: $(env; nSys; sCtrl; plant)^*$

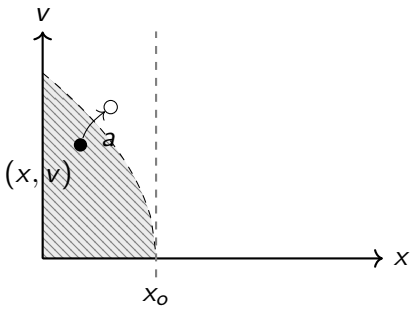


Idea

If $(env; nSys)$ **can** break the invariant, and $plant$ **may** not reestablish the invariant, then safety of $sCtrl$ is properly verified.

Solution

Recall control loop: $(env;\ nSys;\ sCtrl;\ plant)^*$



Idea

If $(env;\ nSys)$ **can** break the invariant, and $plant$ **may** not reestablish the invariant, then safety of $sCtrl$ is properly verified.

Proof Obligation (to be verified in KeYmaera)

$$\exists s.\, \exists e.\, \exists a_1.\, \Big(Inv(s,e,a_1) \wedge \langle env;\ nSys\rangle \Big(\neg Inv(s,e,a)\ \wedge \langle plant\rangle \neg Inv(s,e,a_1)\Big)\Big)$$

Reference

- Yuvaraj Selvaraj, Jonas Krook, Wolfgang Ahrendt, Martin Fabian
  On proving that an unsafe controller is not proven safe
  Journal of Logical and Algebraic Methods in Programming
  Elsevier 2024