Relational Abstract Interpretation for Enforcing Information Flow Security

Seyed Mohammad Mehdi Ahmadpanah

smahmadpanah@aut.ac.ir

Supervisor

Dr. Mehran S. Fallah

Formal Security Lab.

CEIT@AUT

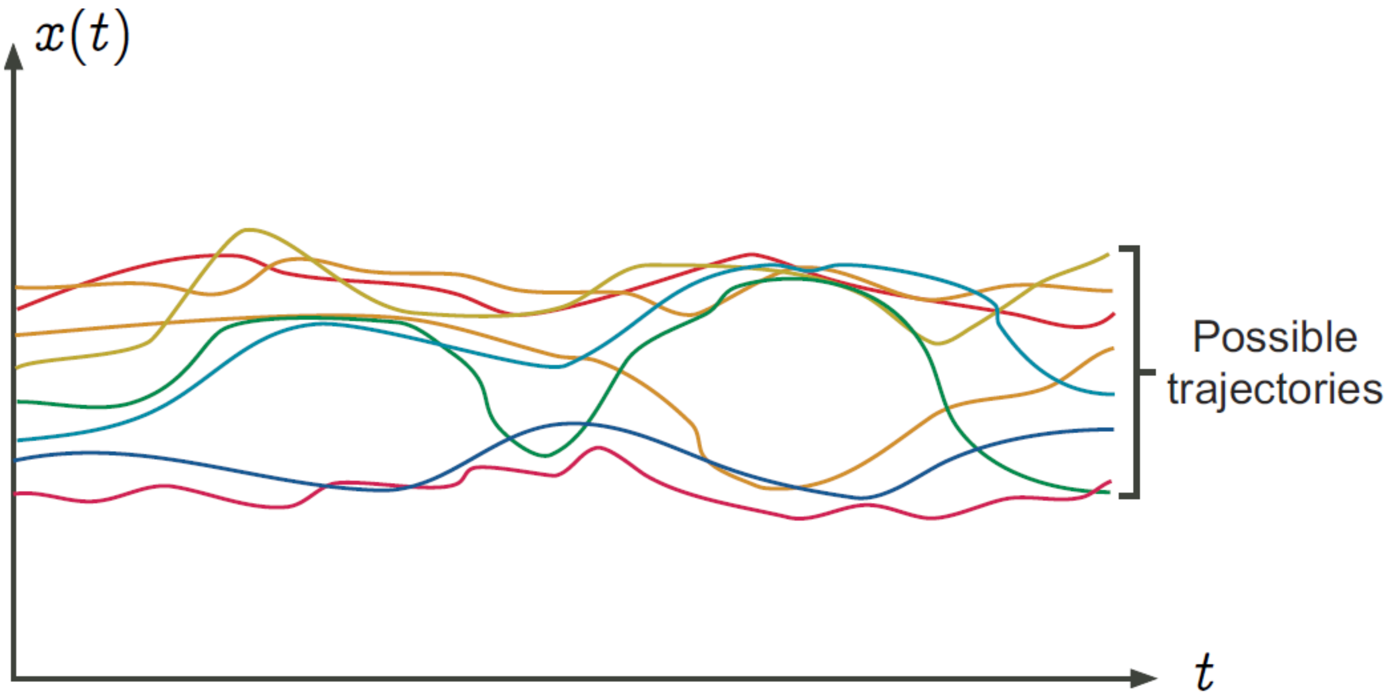
Jul. 31, 2018

Outline

- An Introduction to Abstract Interpretation
  - Abstract Domain
  - Abstract Transform
  - Galois Connection
  - Combination of Galois Connections
- Relational Abstract Interpretation for Verification of 2-hypersafety properties
  - Via self-composition of CFG
  - XML manipulating language

Introduction

- Concrete Semantics
  - formalizes the set of all possible executions of this program in all possible execution environments *(Possible Behaviors)*



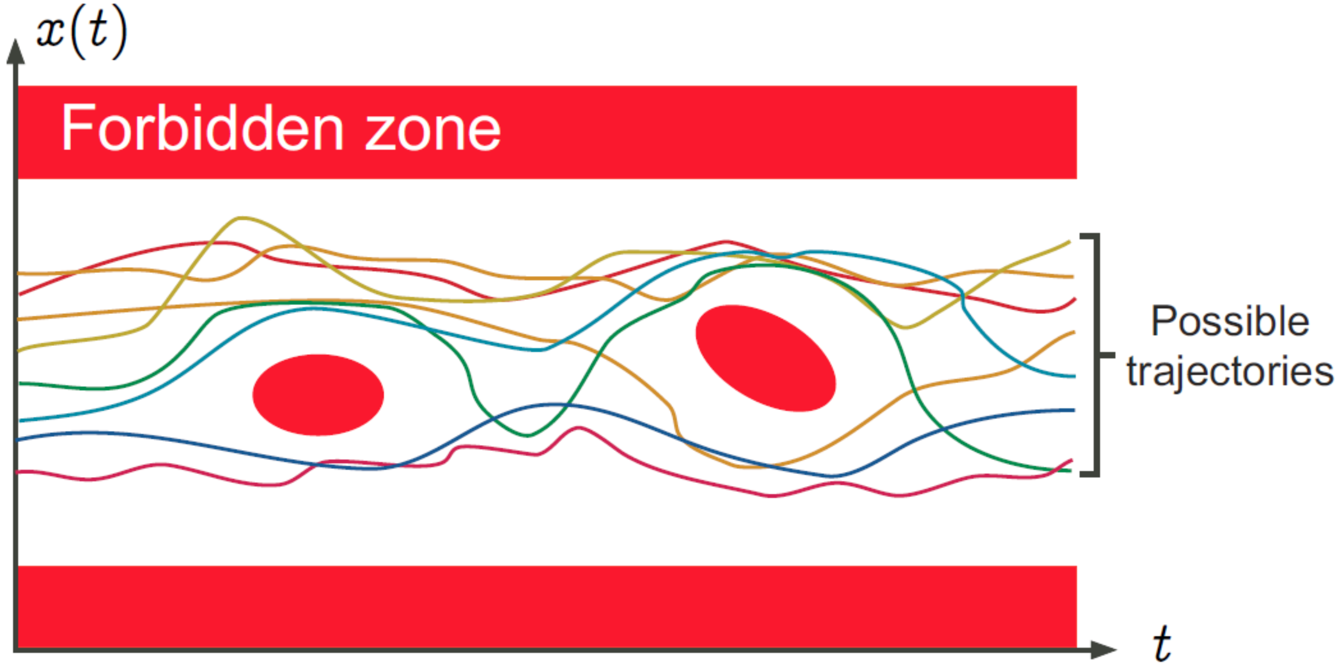
Introduction

- Undecidability
  - The concrete mathematical semantics of a program is an "infinite" mathematical object, not computable;
  - All non trivial questions on the concrete program semantics are undecidable (e.g. termination).
  - Assume termination(P) would always terminates and returns true iff P always terminates on all input data

```
P ≡ while termination(P) do skip od
```

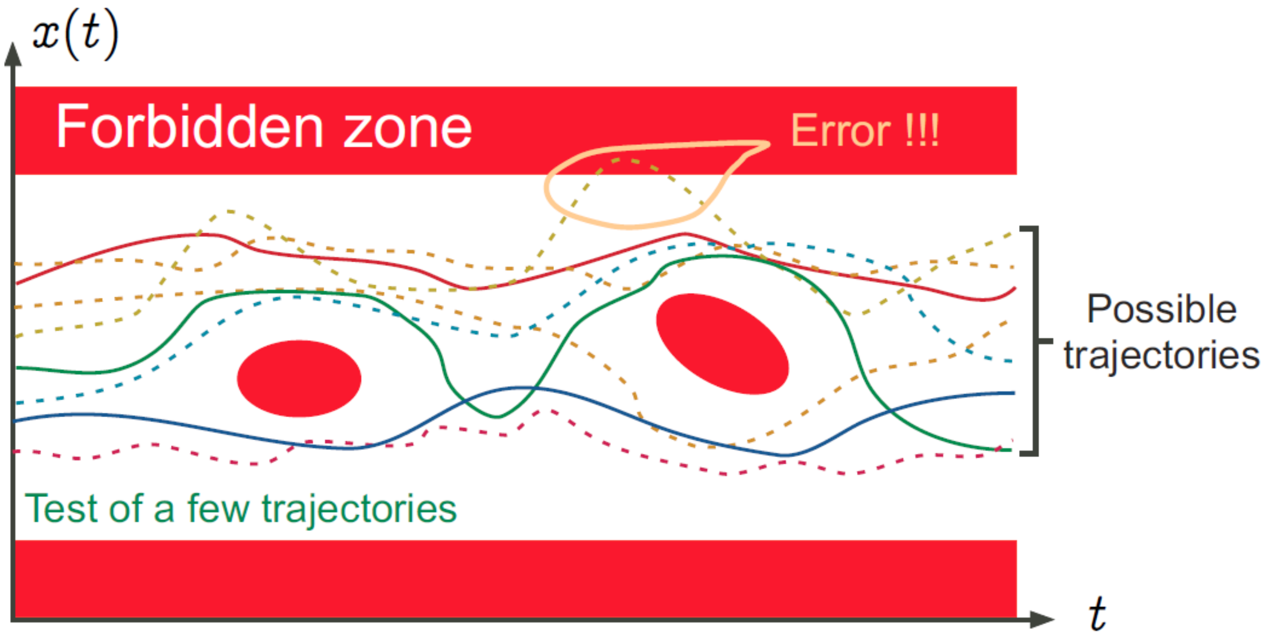
Introduction

- Safety Property
  - expresses that no possible execution of the program when considering all possible execution environments can reach an erroneous state



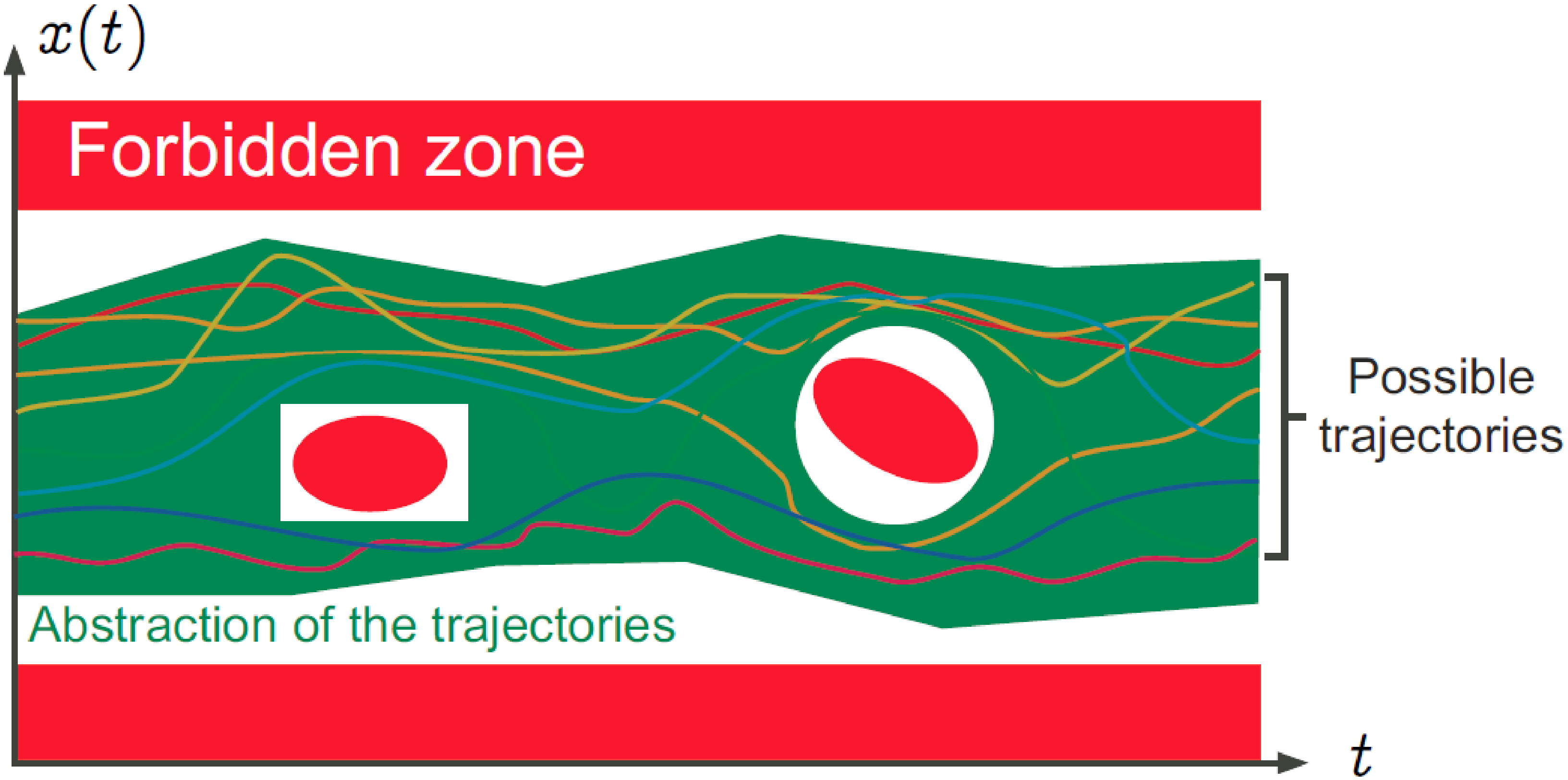
Introduction

- Property Testing/Debugging
  - consists in considering a subset of the possible executions and not a correctness proof;
  - absence of coverage is the main problem



Introduction

- **Abstract Interpretation**



Introduction

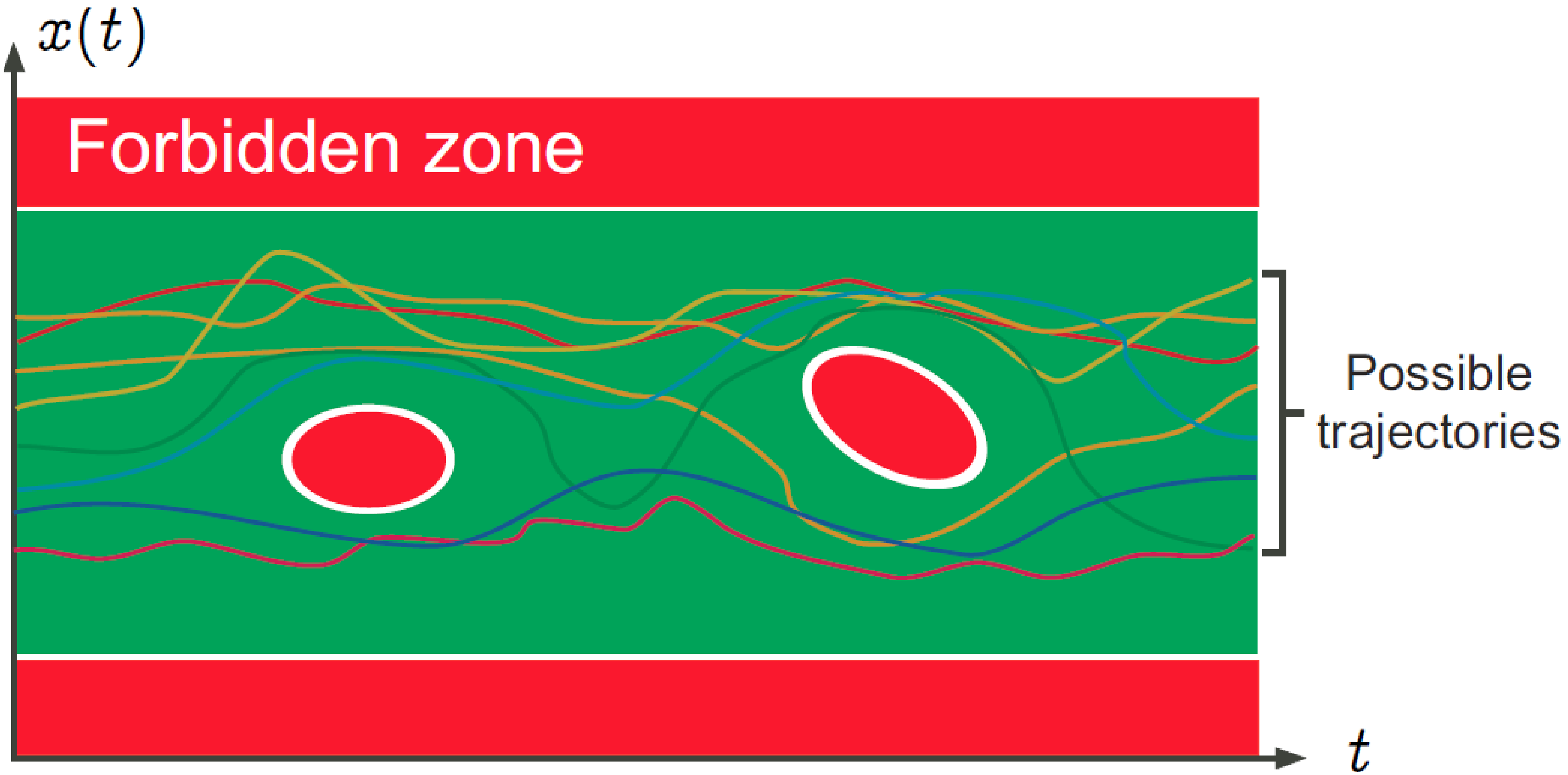
- Formal Methods are Abstract Interpretations!
  - Model Checking: abstract semantics is given manually by the user (a *finitary model* of the program execution)
  - Deductive Methods: the user must provide the abstract semantics in the form of *inductive arguments*
  - Static Analysis: the abstract semantics is computed automatically from the program text according to *predefined abstractions*

Introduction

- Requirements of Abstract Semantics
  - sound so that no possible error can be forgotten
    - no conclusion derived from the abstract semantics is wrong relative to the program concrete semantics and specification
  - precise enough (to avoid false alarms)
  - as simple/abstract as possible

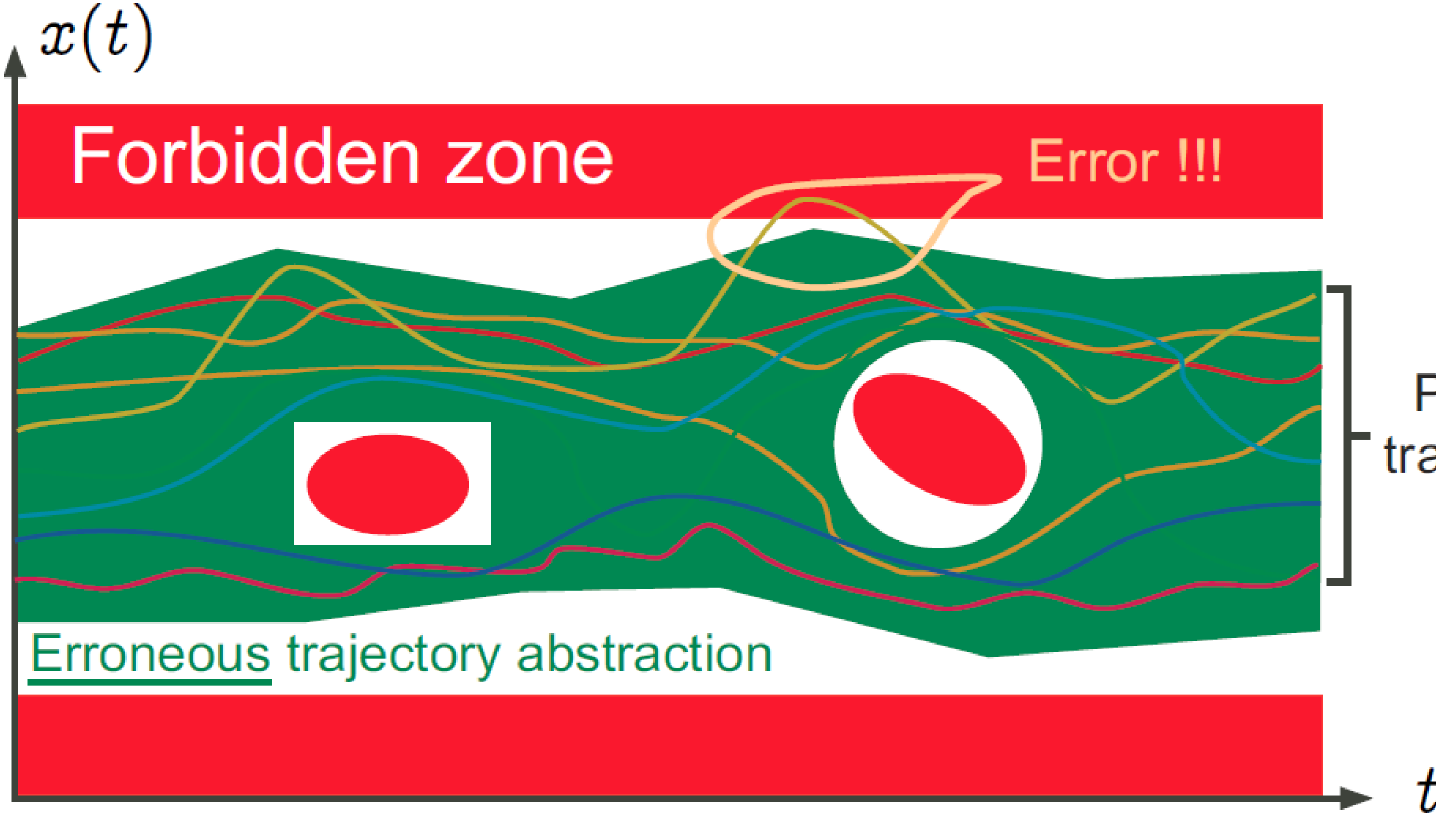
Introduction

- Correct and precise semantics



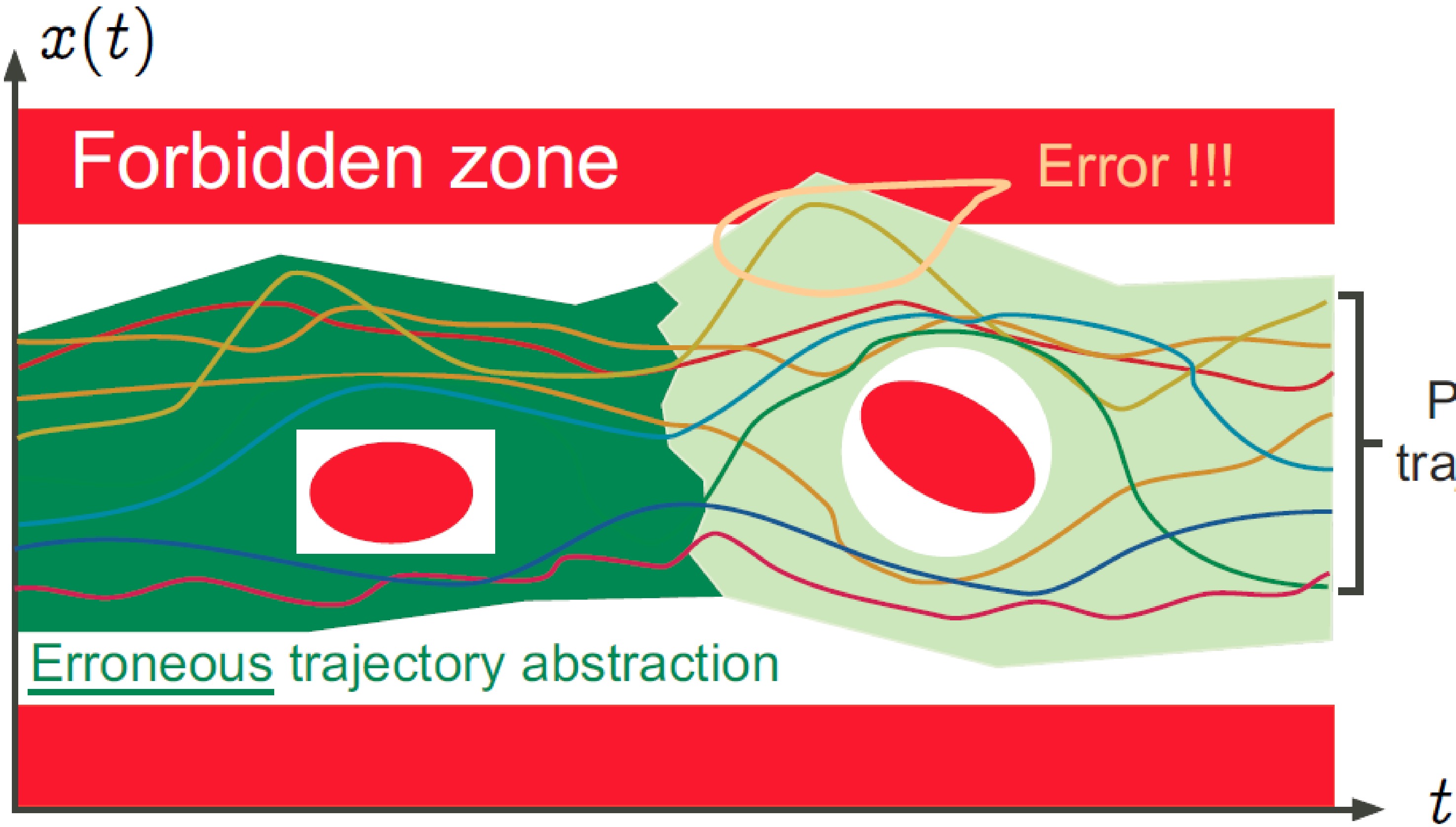
Introduction

- Erroneous semantics



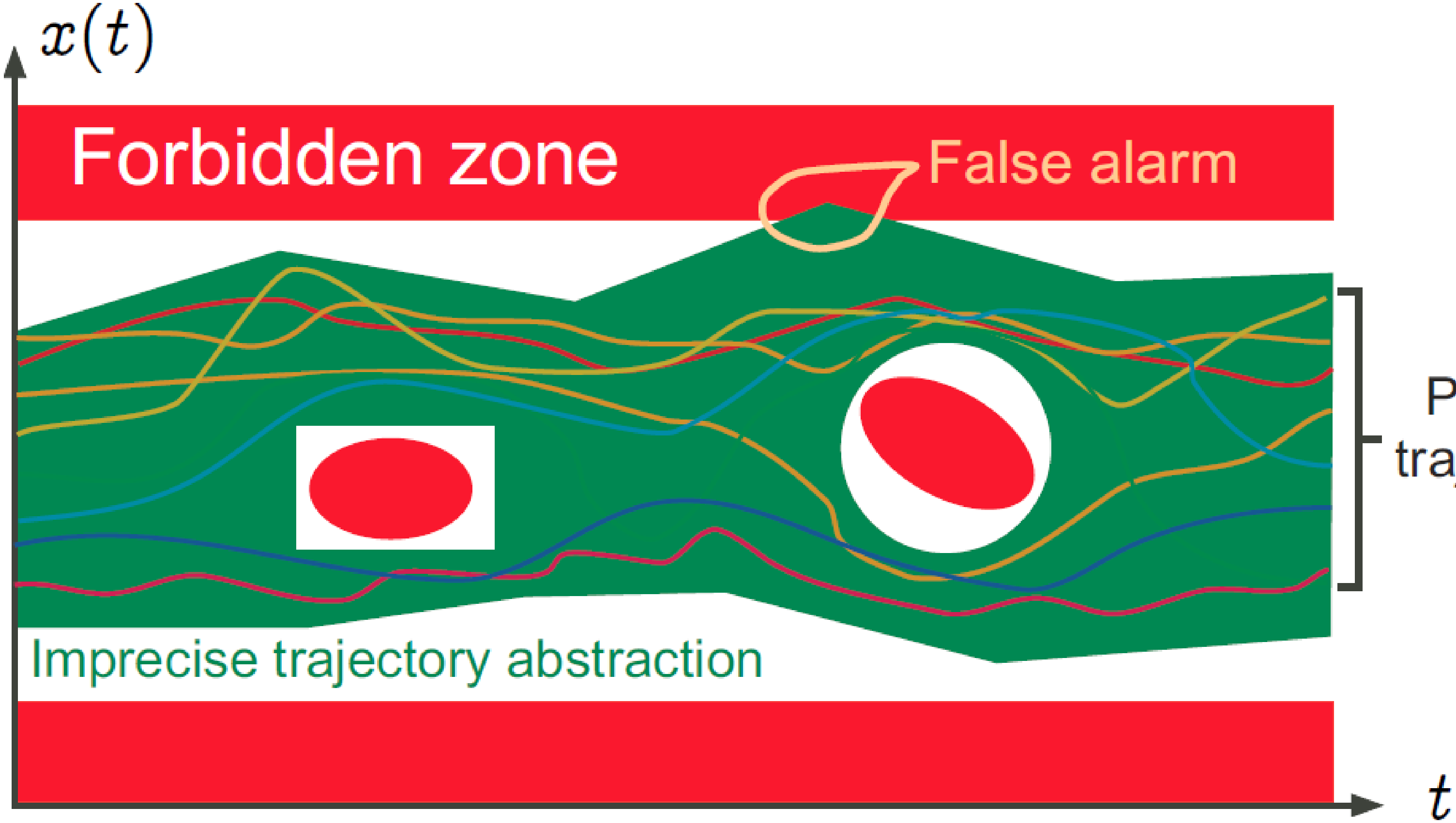
Introduction

- **Another erroneous semantics** (bounded model checking)



Introduction

- **Imprecise semantics (False Alarms)**



Abstract Domain and Transform

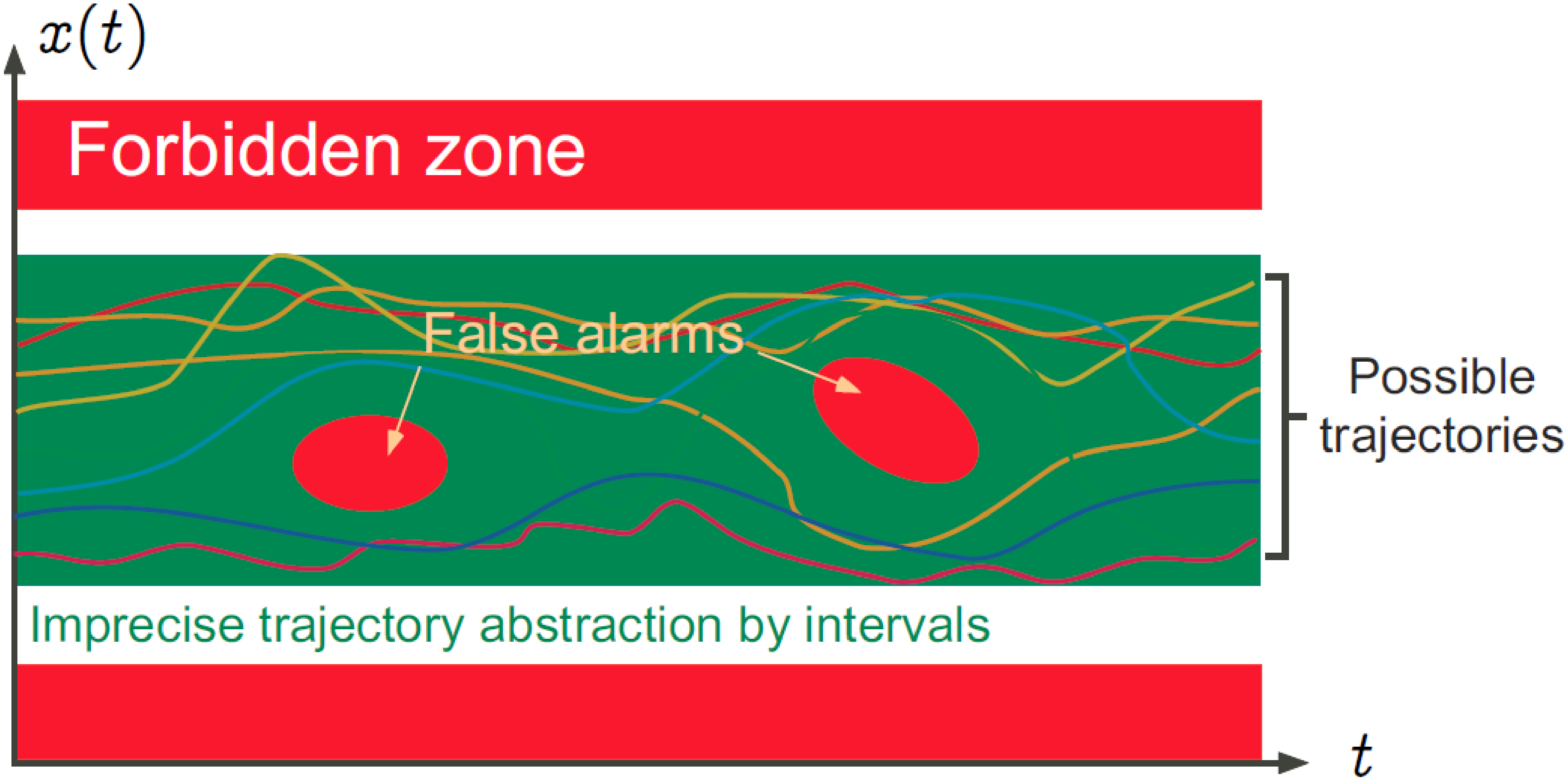
- Abstract Domain
  - providing a description of *abstract program properties* and *abstract property transformers* describing the operational effect of program instructions and commands in the abstract.
- Standard abstractions
  - that serve as a basis for the design of static analyzers
  - abstract program data
  - abstract program basic operations
  - abstract program control (iteration, procedure, concurrency,...)
  - can be parametrized to allow for manual adaptation to the application domains

Abstract Domain and Transform

- Most program properties can be expressed as $fixpoints$ of monotone or extensive property transformers, a property preserved by abstraction.
  - This reduces program analysis to fixpoint approximation and verification to fixpoint checking.

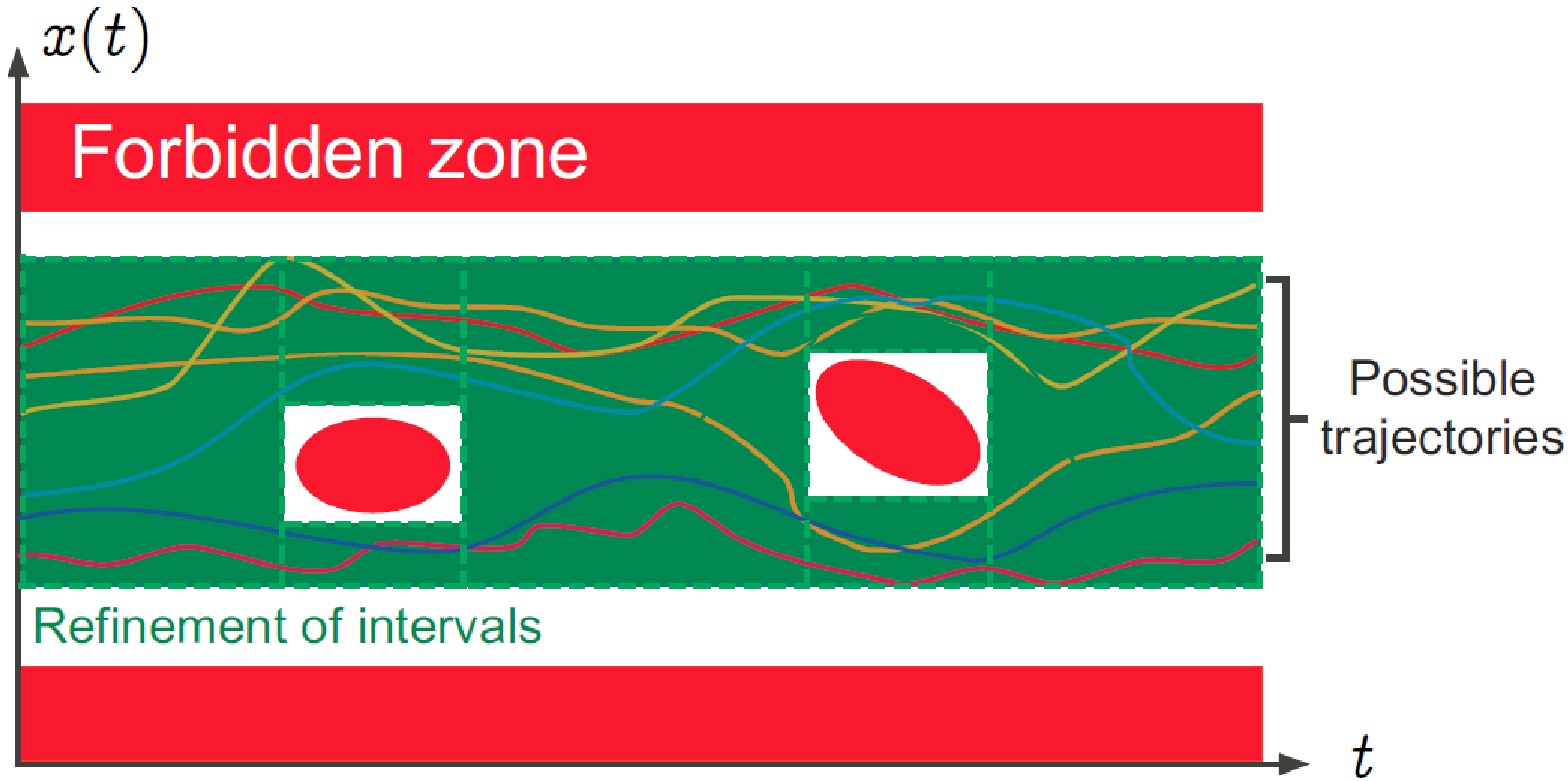
Abstract Domain and Transform

- **Standard abstraction by intervals**



Abstract Domain and Transform

- More refined abstraction



Abstract Domain and Transform

- Approximation Options (Abstract Domains)

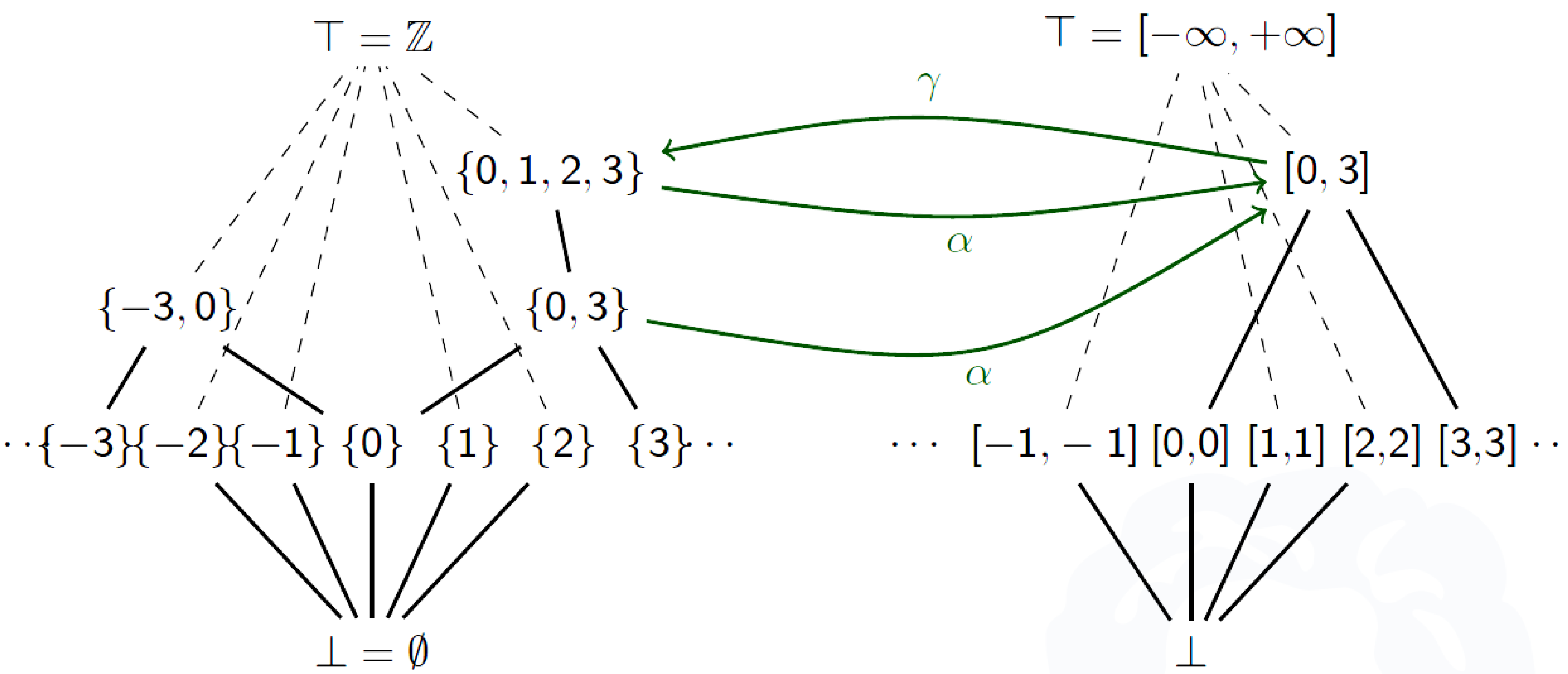
$$
\begin{tikzcd}
V \arrow[r, "\longrightarrow"] \arrow[d, "\beta"'] & V \arrow[d, "\beta"] \\
A \arrow[r, "\triangleright"'] & A
\end{tikzcd}
$$

- $(\longrightarrow) : V \to V$ is operational semantics
- $\beta : V \to A$ is abstraction function
- $\triangleright : A \to A$ is abstraction of $(\longrightarrow)$ wrt A
- $\triangleright$ and $(\longrightarrow)$ must *agree* wrt β

Abstract Domain and Transform

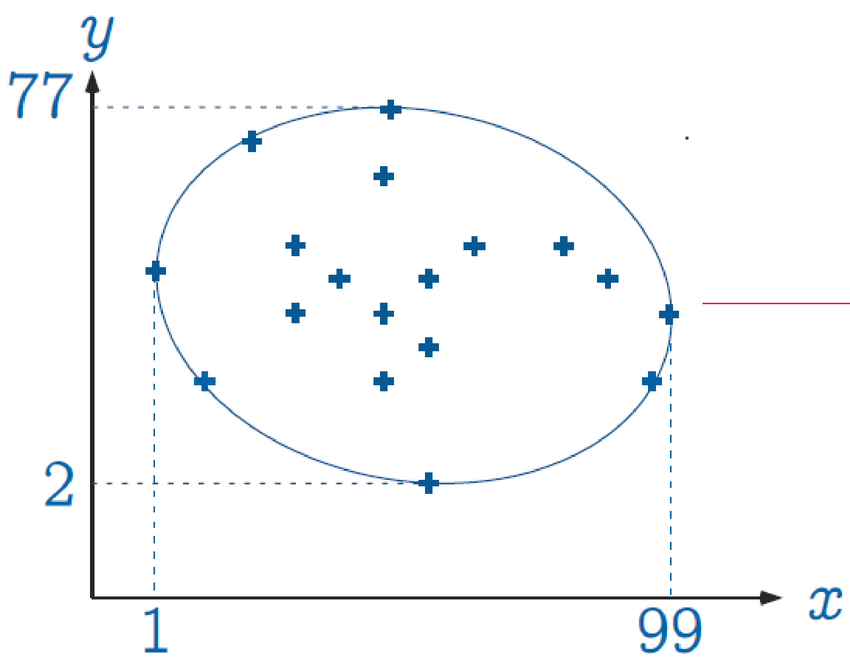
- Choose an abstract domain, replacing sets of objects (states, traces, …) S by their abstraction $\alpha(S)$
- The abstraction function α maps a set of concrete objects to its abstract interpretation
- The inverse concretization function γ maps an abstract set of objects to concrete ones
- $S \subseteq \gamma(\alpha(S))$

Abstract Domain and Transform



Abstract Domain and Transform

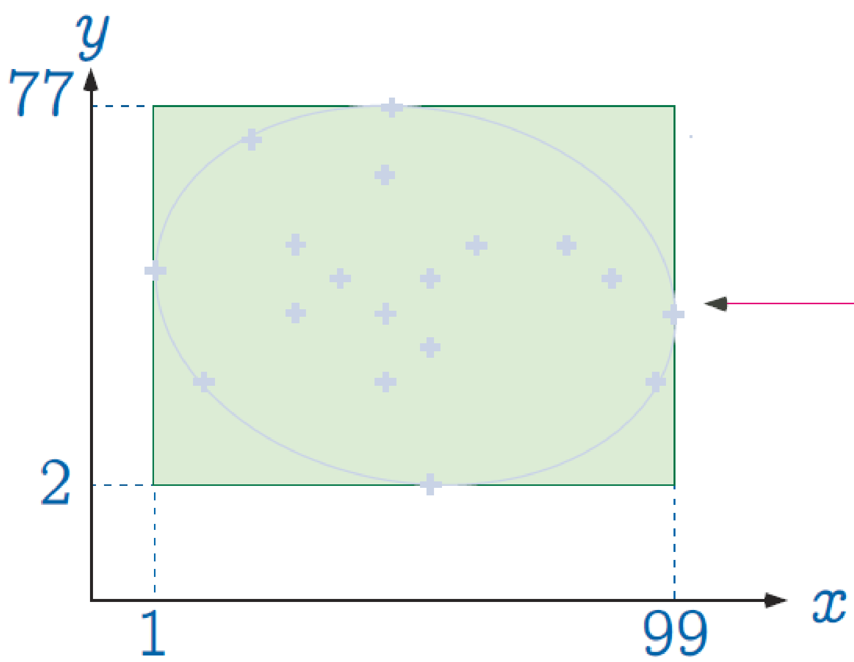
- Interval abstraction α



$$\{x : [1, 99], y : [2, 77]\}$$

Abstract Domain and Transform

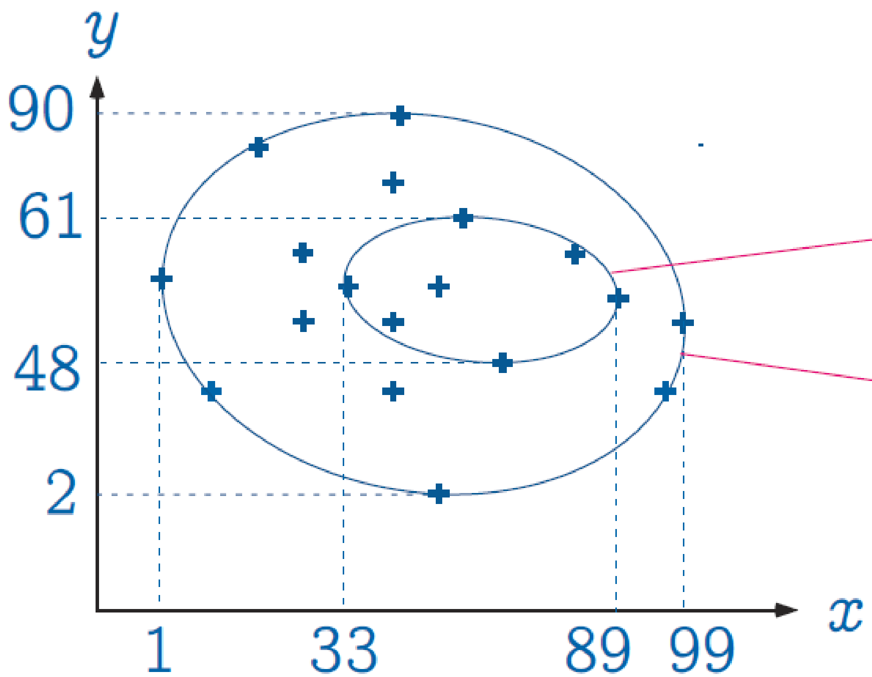
- Interval concretization γ



$\{x : [1, 99], y : [2, 77]\}$

Abstract Domain and Transform

- Abstraction function α is monotone



$\{x : [33, 89], y : [48, 61]\}$

$\sqsubseteq$

$\{x : [1, 99], y : [2, 90]\}$

$$X \subseteq Y \Rightarrow \alpha(X) \sqsubseteq \alpha(Y)$$

Galois Connection

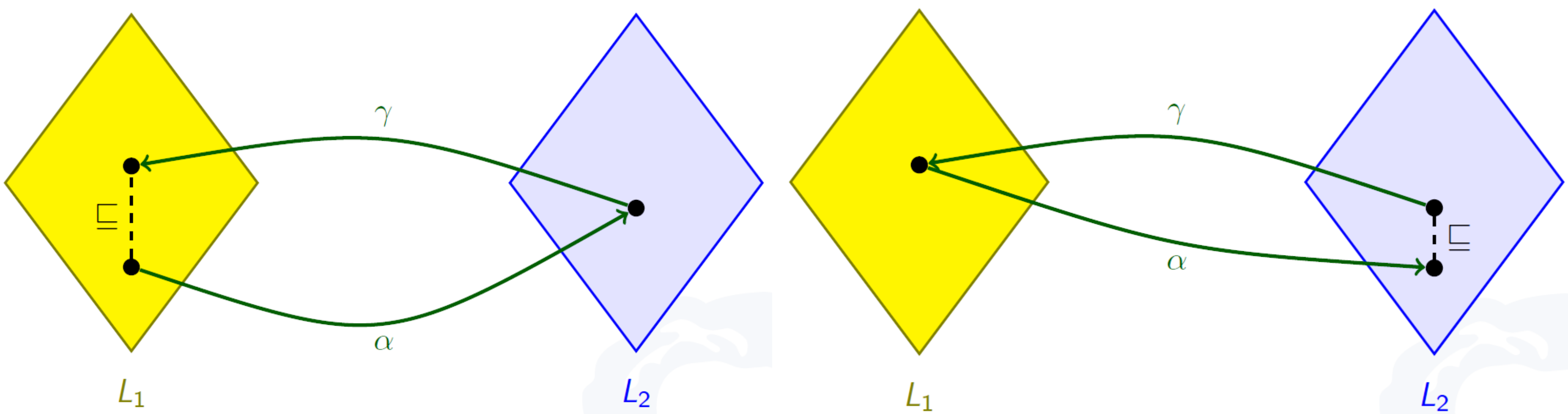
Notation: $L_1 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} L_2$

$\alpha : L_1 \rightarrow L_2$ *monotonic* (i.e., $x \sqsubseteq y \implies \alpha(x) \sqsubseteq \alpha(y)$)

$\gamma : L_2 \rightarrow L_1$ *monotonic*

$y \sqsubseteq \alpha \circ \gamma(y)$

$\gamma \circ \alpha(x) \sqsubseteq x$



Galois Connection

- Galois Insertion = Galois Connection + α is surjective (onto)

$$\gamma\big(\alpha(X)\big) = X$$

  – Common case in program analysis

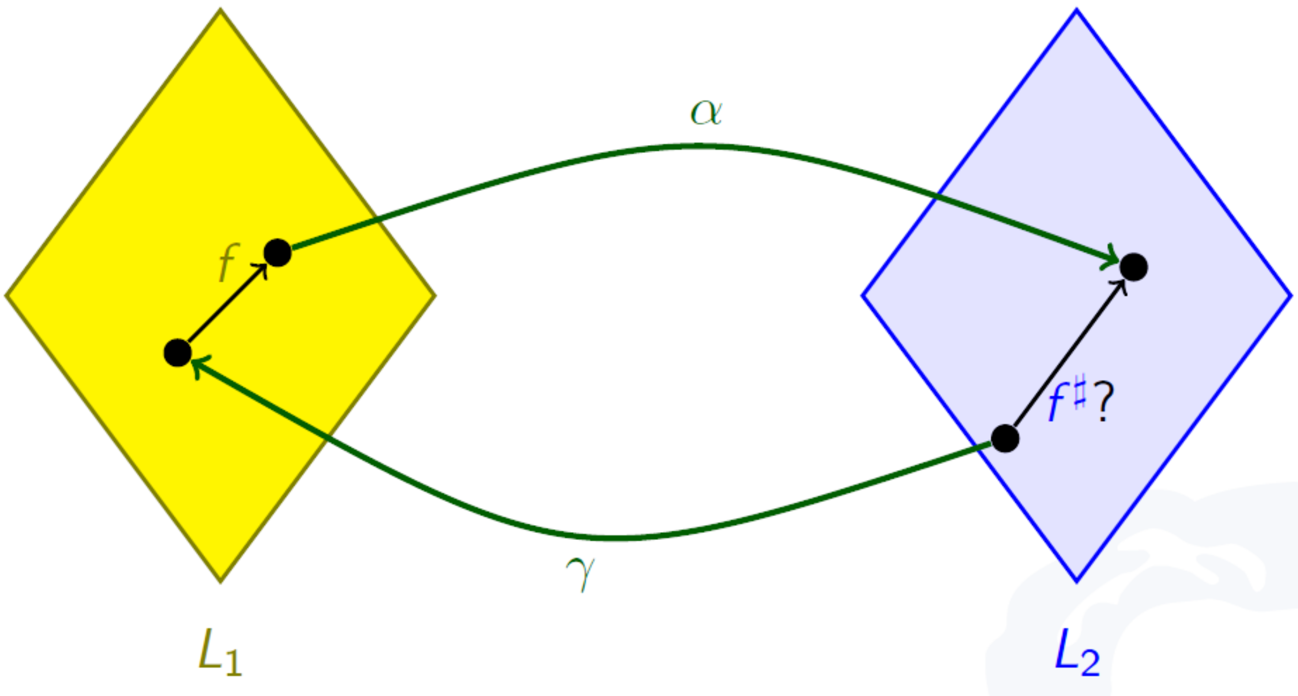
- Constructing Concretization

$$\gamma(y) = \sqcup \{x | \alpha(x) \sqsubseteq y\}$$

Galois Connection

- **Function Abstraction (Induced Operation)**

$$f^{\#} = \alpha \circ f \circ \gamma$$



Combination of Galois Connections

- Sequence

$$L_1 \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} L_2 \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} L_3$$

$$L_1 \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} L_3$$

- Product (Independent Attributes)

$$L_1 \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} M_1$$

$$L_2 \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} M_2$$

$$L_1 \times L_2 \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M_1 \times M_2$$

$$\alpha(\langle x_1, x_2 \rangle) = \langle \alpha_1(x_1), \alpha_2(x_2) \rangle$$
$$\gamma(\langle x_1, x_2 \rangle) = \langle \gamma_1(x_1), \gamma_2(x_2) \rangle$$

Combination of Galois Connections

- **Function**

Galois connection $L \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} M$

Set S

$$S \to L \underset{\alpha_S}{\overset{\gamma_S}{\leftrightarrows}} S \to M$$

$$\alpha_s(f) = \alpha \circ f$$
$$\gamma_s(f) = \gamma \circ f$$

$$\left\{ \begin{array}{ccccccccc} s_1 & \mapsto & l_1 & \xrightarrow{\alpha} & m_1 & \leftarrow\!\shortmid & s_1 \\ s_2 & \mapsto & l_2 & \xrightarrow{\alpha} & m_2 & \leftarrow\!\shortmid & s_2 \\ s_3 & \mapsto & l_3 & \xrightarrow{\alpha} & m_3 & \leftarrow\!\shortmid & s_3 \end{array} \right\}$$

Combination of Galois Connections

- Tensor Product

$$\mathcal{P}(V) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} \mathcal{P}(D_1)$$

$$\mathcal{P}(V) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} \mathcal{P}(D_2)$$

$$\mathcal{P}(V) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{P}(D_1 \times D_2)$$

$$\alpha(v) = \bigcup\{\alpha_1(\{x\}) \times \alpha_2(\{x\}) | x \in v\}$$

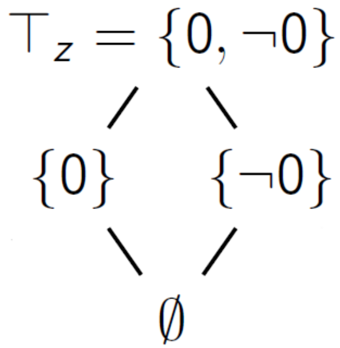
$$\gamma(d_{1,2}) = \{x | \alpha_1(\{x\}) \times \alpha_2(\{x\}) \subseteq d_{1,2}\}$$

Combination of Galois Connections

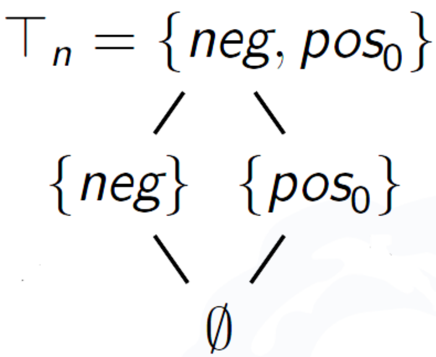
- **Tensor Product**

$$\alpha(\{-2, 0\}) = \{\langle \neg 0, neg\rangle, \langle 0, pos_0\rangle\}$$

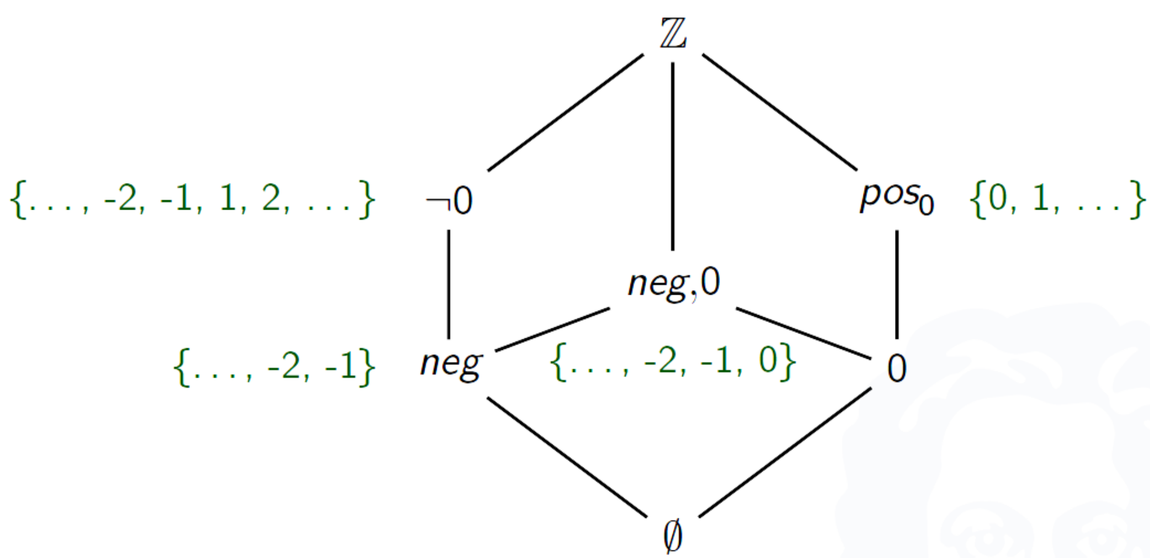
$$\gamma(\{\langle \neg 0, neg\rangle\}) = \{\dots, -1\}$$
$$\gamma(\{\langle 0, pos_0\rangle\}) = \{0\}$$
$$\gamma(\{\langle \neg 0, neg\rangle, \langle 0, pos_0\rangle\}) = \{\dots, -1, 0\}$$



L_z



L_n



Relational Abstract Interpretation for the Verification of 2-Hypersafety Properties

Motivation

- Abstract interpretation is a well established approach for proving $safety$ properties of programs
- Information Flow policies are formalized as $safety$ $hyperproperties$
  - Properties of multiple runs
- The verification of k-hypersafety properties can be reduced to the verification of ordinary safety properties of the k-fold self-composition of the program

Motivation

- A
  -
- Nc
  - ic
    et
- Ta ting
  lar
  - je

```
<if name="If">
 <condition> <![CDATA[$test < 0.5]]>
 </condition>
  <assign name="EvalGood">
   <copy> <from>"good"</from>
    <to> $pList/patientRecord[id=$patientId]
                /health/text()
    </to> </copy>
  </assign>
 <else>
  <assign name="EvalPoor">
   <copy> <from>"poor"</from>
    <to> $pList/patientRecord[id=$patientId]
                /health/text()
    </to> </copy>
  </assign>
 </else>
</if>
```

Set up

- Execution $=$ sequence of assignments and condition evaluations
- Inserting `skip` instructions into arbitrary places of an execution does not change the result
  - Different $alignments$ of a pair of executions
- Knowing an initial abstract value (the potential set of pairs of initial states), computing the final abstract value (the possible set of pairs of states that can result from any pairs of executions)

Main Idea

- Relational abstract domain describing *pairs* of concrete states + abstract transformers for *pairs* of instructions set in alignment within the two executions
- The abstract effect of a given pair of executions w.r.t a fixed alignments of instructions
  - By applying the composition of the occurring transformers to the initial abstract value
- Abstract effect of any alignment results in a safe overapproximation of the desired outcome
  - Approximate it with the *glb* over all alignments

Main Idea

- To obtain a safe approximation for all pairs of executions reaching a given pair of program points
  - lub of the values provided for each pair of executions individually
  - Merge over all Twin Computations (MTC) solution
- To achieve maximal precision, the structural similarities of two subprograms are taken into account using a tree distance measure during the construction.
- Selecting one promising static alignment of instructions for each pair of executions resulting in a $self\text{-}composition\ of\ the\ CFG$ of the program.

Merge Over All Twin Computations

- the semantics of programs defined by means of control-flow graphs (CFG)

$$G = (N, E, n_{in}, n_{fi})$$

- N: set of nodes
- n_{in}, n_{fi}: unique initial and final nodes
- E: set of directed and labeled edges

$$(n_1, f, n_2)$$

f: state transformer

$$[\![f]\!] : S \nrightarrow S$$

Merge Over All Twin Computations

- An execution = A path from the initial node to the final node
- The effect of the sequence of labels $\pi = f_1 \ldots f_n$,

$$[\![\pi]\!]s_0 = [\![f_1 \ldots f_n]\!]s_0 = [\![f_n]\!] \circ \ldots \circ [\![f_1]\!]s_0$$

$n_1 \rightsquigarrow n_2$: the set of sequences of labels of paths from node n_1 to node n_2

Merge Over All Twin Computations

- We define a 2-hypersafety property by an initial and a final relation of program states $\rho_{in}, \rho_{fi} \subseteq S \times S$
- A program given by CFG $G = (N, E, n_{in}, n_{fi})$ satisfies the 2-hypersafety property specified by in ρ_{in} and ρ_{fi}, if for all pairs of initial states $(s_0, t_0) \in \rho_{in}$ and all pairs of final states $s = [\![\pi_1]\!]s_0$ and $t = [\![\pi_2]\!]t_0$ reachable by arbitrary computations $\pi_1, \pi_2 \in n_{in} \rightsquigarrow n_{fi}$, it holds that $(s, t) \in \rho_{fi}$.

Merge Over All Twin Computations

- $(\mathbb{D}, \sqsubseteq)$ complete lattice
- $(\mathcal{P}(S \times S), \alpha, \gamma, \mathbb{D})$ Galois connection
  - Powerset of pairs of states $\mathcal{P}(S \times S)$
  - Abstraction function $\alpha{:}\mathcal{P}(S \times S) \to \mathbb{D}$
  - Concretization function $\gamma{:}\ \mathbb{D} \to \mathcal{P}(S \times S)$
- Requirement for abstract transformers $[\![f, g]\!]^{\#}$ of pairs of labels (f, g) (called twin steps)

$$\gamma([\![f, g]\!]^{\sharp} d) \quad \supseteq \quad \{ \quad (s', t') \mid \exists (s, t) \in \gamma(d) : [\![f]\!]s = s' \wedge [\![g]\!]t = t' \quad \}$$

Merge Over All Twin Computations

- The set of all possible alignments $A(\pi_1, \pi_2)$ is recursively defined by

$$
\begin{array}{lcl}
A(\varepsilon, \varepsilon) & = & \varepsilon \cup \{(\texttt{skip}, \texttt{skip})\omega \mid \omega \in A(\varepsilon, \varepsilon)\} \\
A(\varepsilon, g\pi) & = & \{(\texttt{skip}, g)\omega \mid \omega \in A(\varepsilon, \pi)\} \cup \\
 & & \{(\texttt{skip}, \texttt{skip})\omega \mid \omega \in A(\varepsilon, g\pi)\} \\
A(f\pi, \varepsilon) & = & \{(f, \texttt{skip})\omega \mid \omega \in A(\pi, \varepsilon)\} \cup \\
 & & \{(\texttt{skip}, \texttt{skip})\omega \mid \omega \in A(f\pi, \varepsilon)\} \\
A(f\pi_1, g\pi_2) & = & \{(f, g)\omega \mid \omega \in A(\pi_1, \pi_2)\} \cup \\
 & & \{(\texttt{skip}, g)\omega \mid \omega \in A(f\pi_1, \pi_2)\} \cup \\
 & & \{(f, \texttt{skip})\omega \mid \omega \in A(\pi_1, g\pi_2)\} \cup \\
 & & \{(\texttt{skip}, \texttt{skip})\omega \mid \omega \in A(f\pi_1, g\pi_2)\}
\end{array}
$$

Merge Over All Twin Computations

- An alignment of two sequences of labels of a CFG, π_1 and π_2, is a sequence of twin steps ω representing both of the original runs
- If $s = [\![\pi_1]\!]s_0$, $t = [\![\pi_2]\!]t_0$, $\omega \in A(\pi_1, \pi_2)$ and $\omega = (f_1, g_1) \dots (f_n, g_n)$ then $(s, t) = [\![\omega]\!](s_0, t_0)$
- For an abstract value d_0, <u>the most precise</u> abstract value d that can be composed using the abstract semantics of twin steps:

$$d = \bigsqcap_{\omega \in A(\pi_1, \pi_2)} [\![\omega]\!]^\sharp d_0$$

Merge Over All Twin Computations

- Given a CFG $G = (N, E, n_{in}, n_{fi})$ and the initial abstract value d_0, the merge over all twin computations solution is defined by:

$$MTC(G, d_0) = \bigsqcup_{\substack{\pi_1 \in n_{in} \rightsquigarrow n_{fi} \\ \pi_2 \in n_{in} \rightsquigarrow n_{fi}}} \bigsqcap_{\omega \in A(\pi_1, \pi_2)} [\![\omega]\!]^{\sharp} d_0$$

  - The MTC solution can be considered as the extension of the MOP (meet over all paths) solution
- Theorem -

$$d \sqsupseteq MTC(G, d_0) \;\; implies \; (s, t) \in \gamma(d)$$

Merge Over All Twin Computations

- The MTC solution is an abstraction of all possible pairs of states resulting from any pair of executions of the program.
- It might be difficult to compute the MTC solution directly.
- A fixed alignment for each pair of paths obtained by constructing a self-composition GG of the CFG G.
  - Perhaps less precise but still sound solution

Merge Over All Twin Computations

- Given the CFG $G = (N, E, n_{in}, n_{fi})$ and a self-composition of it $\mathrm{G}G = (N', E', n'_{in}, n'_{fi})$, the following holds for all d_0:

$$\bigsqcup_{\omega \in n'_{in} \rightsquigarrow n'_{fi}} [\![\omega]\!]^{\sharp} d_0 \sqsupseteq MTC(G, d_0)$$

  – Any solution of the analysis problem corresponding to the self-composition GG of G is a safe overapproximation of MTC(G, d_0).

Verification

- Proving a 2-hypersafety property by means of this methods succeeds in two steps:

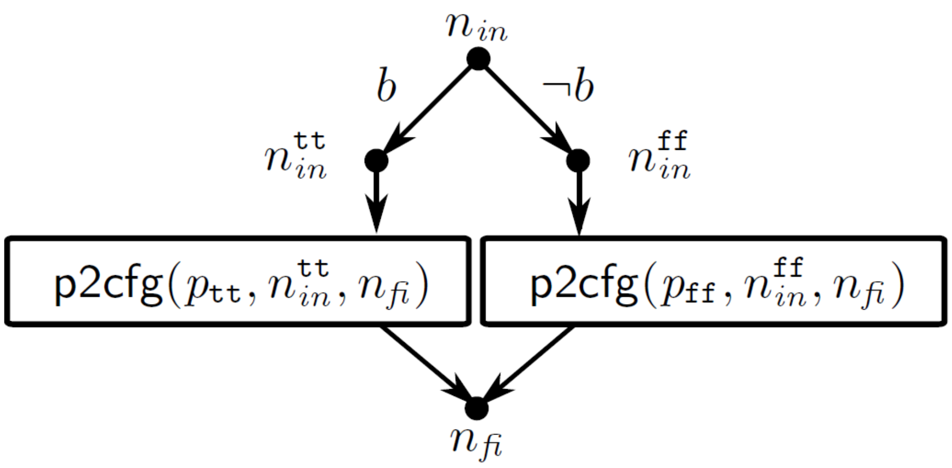
  1- Constructing a self-composition GG of the CFG

  2- Finding a decent abstract domain which achieve reasonable precision at an acceptable price

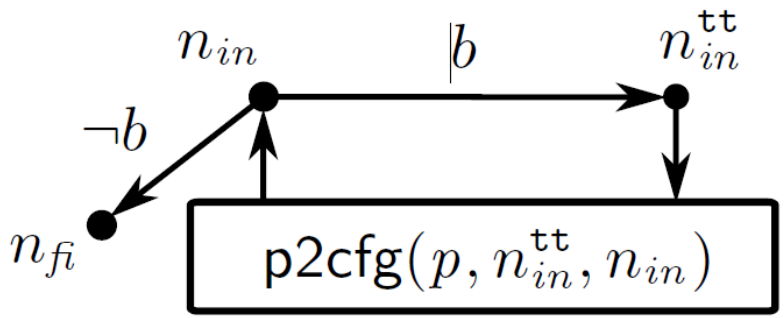
Self-Composition of CFG

$$\begin{array}{llll} (\text{program}) & p & ::= & c; \mid c; p \\ (\text{command}) & c & ::= & \texttt{skip} \mid x \texttt{ := } e \mid \texttt{while}\ b\ \{p\} \mid \\ & & & \texttt{if}\ b\ \{p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\} \end{array}$$

c2cfg($\texttt{if}\ b\ \{\ p_{\texttt{tt}}\}\ \texttt{else}\ \{p_{\texttt{ff}}\}, n_{in}, n_{fi}$):



c2cfg($\texttt{while}\ b\ \{p\}, n_{in}, n_{fi}$):



pp2cfg (pair of programs to CFG) and pc2cfg (pair of commands to CFG)

Self-Composition of CFG

- **Computing a Best Alignment of Two Programs**

$$\omega_{\text{opt}} = \underset{\omega \in A(p_1, p_2)}{\arg\min} \sum_{1 \leq i \leq |\omega|} \text{td}(\omega[i].1, \omega[i].2)$$

- **Computing the Compositions of CFGs of Pairs of Commands**

$$\omega = (c_1, d_1) \ldots (c_k, d_k)$$

Self-Composition of CFG

- The roots of the ASTs corresponding to c and d are considered composable in the following cases:

$$c = d = x := e \text{ or } c = d = \texttt{skip}$$

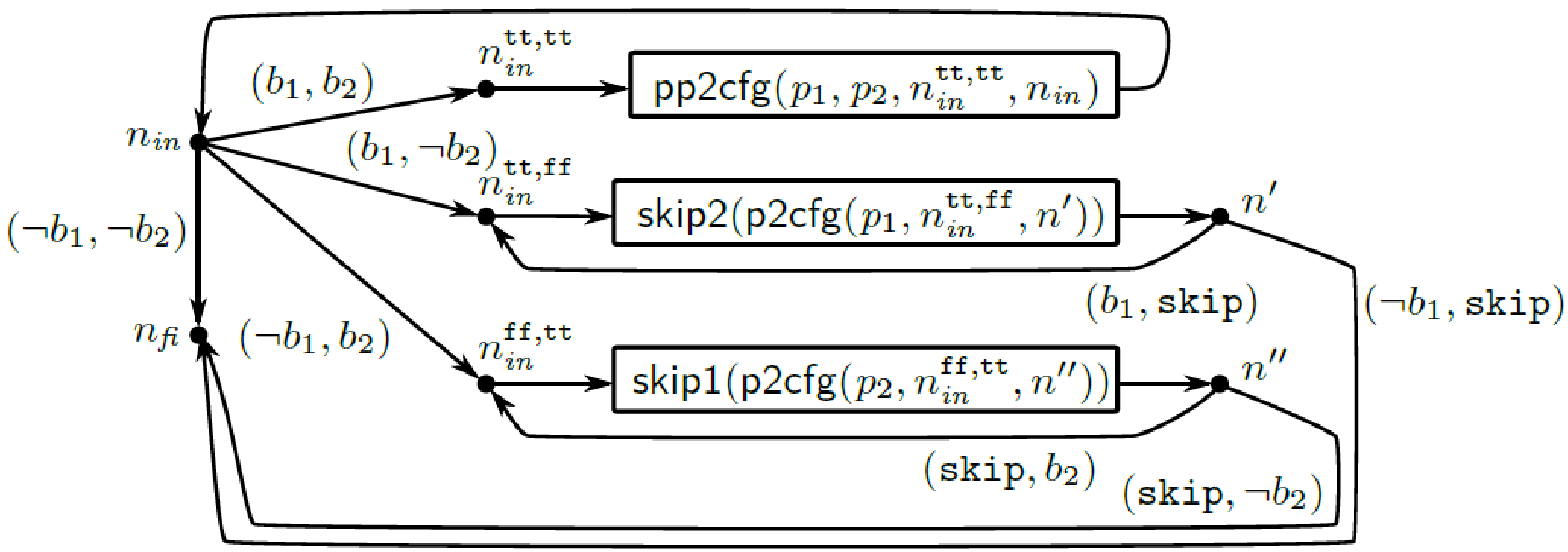
$$c = \texttt{if } b_1 \ \{p^1_{\texttt{tt}}\} \texttt{ else } \{p^1_{\texttt{ff}}\} \text{ and } d = \texttt{if } b_2 \ \{p^2_{\texttt{tt}}\} \texttt{ else } \{p^2_{\texttt{ff}}\}$$

$$c = \texttt{while } b_1 \ \{p_1\} \text{ and } d = \texttt{while } b_2 \ \{p_2\}$$

- In case the roots of the ASTs of the commands c and d are not composable, then we put them in sequence
  - Adding `skip` to each CFG labels

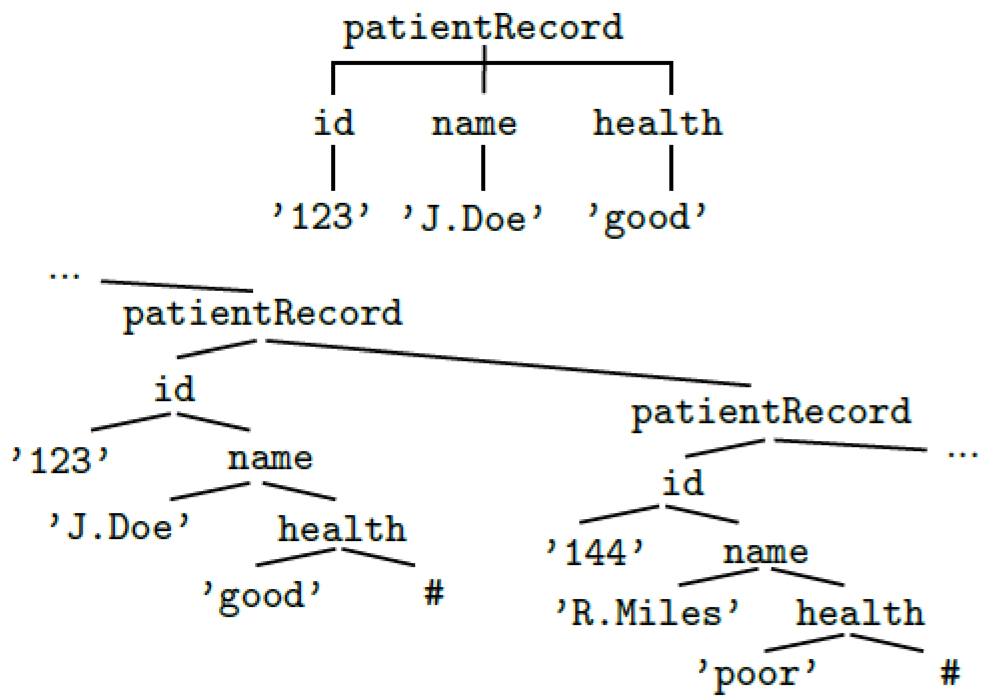
Self-Composition of CFG

- For composable roots
  - Not branching constructs: pc2cfg$(c, d, n_{in}, n_{fi}) = (n_{in}, (c, d), n_{fi})$
  - Branching constructs as follow



Proving Noninterference

- XML documents are unranked ordered trees
  - Represented using binary trees by means of first-child-next-sibling



Proving Noninterference

- Σ_2 : alphabet of binary nodes (models XML tags)
- Σ_0 : alphabet of nullary nodes (models textual entities)
- $\mathcal{T}_{\Sigma_2,\Sigma_0}$: set of binary trees

$$
\begin{array}{llll}
\text{(tree expressions)} & e & ::= & \texttt{\#} \mid x \mid x\texttt{/1} \mid x\texttt{/2} \mid \\
& & & \sigma_2(x, y) \mid \\
& & & \lambda_t(x_1, x_2, \ldots) \\
\text{(Boolean expressions)} & b & ::= & \texttt{top}(x)\texttt{=}\sigma \mid \\
& & & \lambda_b(x_1, x_2, \ldots)
\end{array}
$$

State transformers of edges (concrete transformers)

$$
\begin{aligned}
[\![\texttt{top}(x)\texttt{=}\sigma]\!]s &= s \text{ if } \sigma \in \Sigma_2 \cup \{\texttt{\#}\} \text{ and} \\
&\quad\; s(x) \text{ has root labeled with } \sigma \\
[\![\neg\texttt{top}(x)\texttt{=}\sigma]\!]s &= s \text{ if } \sigma \in \Sigma_2 \cup \{\texttt{\#}\} \text{ and} \\
&\quad\; s(x) \text{ has root labeled with } \sigma' \neq \sigma \\
[\![\lambda_b(x_1, x_2, \ldots)]\!]s &= s \text{ if } [\![\lambda_b]\!](s(x_1), s(x_2), \ldots) \text{ holds}
\end{aligned}
$$

$$
[\![x\texttt{:=}y]\!]s = s[x \mapsto s(y)] \qquad [\![x\texttt{:=\#}]\!]s = s[x \mapsto \texttt{\#}]
$$

$$
[\![x\texttt{:=}\sigma_2(x_1, x_2)]\!]s = s[x \mapsto \sigma_2(s(x_1), s(x_2))]
$$

$$
[\![x\texttt{:=}y\texttt{/1}]\!]s = \begin{cases} s[x \mapsto t_1] & \text{if } s(y) = \sigma_2(t_1, t_2) \text{ for some} \\ & \text{label } \sigma_2\text{, and trees } t_1 \text{ and } t_2 \\ \lightning & \text{otherwise} \end{cases}
$$

$$
[\![x\texttt{:=}y\texttt{/2}]\!]s = \begin{cases} s[x \mapsto t_2] & \text{if } s(y) = \sigma_2(t_1, t_2) \text{ for some} \\ & \text{label } \sigma_2\text{, and trees } t_1 \text{ and } t_2 \\ \lightning & \text{otherwise} \end{cases}
$$

$$
[\![x\texttt{:=}\lambda_t(x_1, x_2, \ldots)]\!]s = s[x \mapsto [\![\lambda_t]\!](s(x_1), s(x_2), \ldots)] \text{ or } \lightning
$$

$$
[\![f]\!]\lightning = \lightning \text{ for all edges } f
$$

Proving Noninterference

- Termination-Insensitive Noninterference

  $$if\ \ s_0|_L = t_0|_L\ then\ \ s|_L = t|_L\ \ for\ all\ pairs\ of\ executions$$

- Abstract states: $d: Var \rightarrow \mathcal{P}(\mathcal{T}_{\Sigma_2,\{\#,bv,*\}})$
- $(s,t) \in \gamma(d)\ \ if\ \ for\ all\ variables\ x, \big(s(x), t(x)\big) \in \gamma(d(x))$
- A pair of trees $\mathcal{T}_1, \mathcal{T}_2$ is in the concretization of a set Λ of abstract trees, if Λ contains a tree $\mathcal{T}$ such that both $\mathcal{T}_1$ and $\mathcal{T}_2$ can be obtained from $\mathcal{T}$ by replacing the occurrences of bv with identical basic values, and the occurrences of $*$ with any, possibly different subtrees.

Example - $\Lambda = \{a(*, bv), b(bv,*)\}$

$\mathcal{T}_1, \mathcal{T}_2 \in \gamma(\Lambda)$ for $\mathcal{T}_1 =$ `a('secret','2')` and $\mathcal{T}_2 =$ `a('SECRET','2')`

Proving Noninterference

- Abstract values do not record sensitive data and precise information on basic values
- $var_{x,n}$: sets of public views for variable x occurring at a node n
  - Defined by means of Horn clauses
    - Disjunction of literals with at most one positive literal
- $\mathcal{T} \in d(x)\ at\ node\ n\ if\ var_{x,n}(\mathcal{T})\ holds$

Proving Noninterference

- **Assignments as Horn clauses**

  Transformers of the form $[\![x \coloneqq e_1(x_1, .. x_n),\ y \coloneqq e_2(y_1, \dots, y_n)]\!]$

  - For all variables $z \neq x\ \ and\ z \neq y$: $var_{z,n'}(X) \Leftarrow var_{z,n}(X)$
  - For edges with label $(x \coloneqq y, x \coloneqq y)$: $var_{x,n'}(X) \Leftarrow var_{y,n}(X)$
  - For edges with label $(x \coloneqq \sigma_2(y,z), x \coloneqq \sigma_2(y,z))$:

    $$var_{x,n'}(\sigma_2(L,R)) \Leftarrow var_{y,n}(L), var_{z,n}(R)$$

  - For edges with label $(x \coloneqq y/1, x \coloneqq y/1)$:

    $$var_{x,n'}(L) \Leftarrow var_{y,n}\big(\sigma_2(L,\_)\big)\ for\ all\ \sigma_2$$

  - For edges with label $(x \coloneqq y/2, x \coloneqq y/2)$:

    $$var_{x,n'}(R) \Leftarrow var_{y,n}\big(\sigma_2(\_,R)\big)\ for\ all\ \sigma_2$$

Proving Noninterference

- **Assignments as Horn clauses**
  - For edges with label (f, f) where $f = x \coloneqq \lambda_t(x_1, \ldots, x_k)$

$$\begin{aligned}\mathtt{var}_{x,n'}(bv) \quad \Leftarrow \quad & \mathtt{var}_{x_1,n}(\_),\mathtt{var}_{x_2,n}(\_),\\ & \ldots,\mathtt{var}_{x_k,n}(\_).\end{aligned}$$

$$\begin{aligned}\mathtt{var}_{x,n'}(\star) \quad \Leftarrow \quad & \mathtt{var}_{x_i,n}(\mathtt{X}),\mathtt{secret}(\mathtt{X}),\\ & \mathtt{var}_{x_1,n}(\_),\ldots,\\ & \mathtt{var}_{x_k,n}(\_).\end{aligned}$$

  - For edges with label $(x \coloneqq e(x_1 \ldots x_k), skip)$ and $(skip, x \coloneqq e(x_1 \ldots x_k))$

$$\mathtt{var}_{x,n'}(\star) \Leftarrow \mathtt{var}_{x_1,n}(\_),\ldots,\mathtt{var}_{x_k,n}(\_)$$

- **Boolean Expressions as Horn clauses (similarly)**

Proving Noninterference

- Theorem - The abstract transformer $[\![f,g]\!]^{\#}$ for a pair (f,g) as defined by Horn clauses is a correct abstract transformer.
  - In other words, if $[\![f]\!](\mathrm{s}_0) = s$ and $[\![g]\!](\mathrm{t}_0) = \mathrm{t}$ where $(s_0, t_0) \in \gamma(d_0)$ and $[\![f,g]\!]^{\#} d_0 = d$, then $(s,t) \in \gamma(d)$ holds.
- The least solution of the set of Horn clauses defined for a program over-approximates the MTC solution.
- Noninterference for a particular output variable x holds at program exit n_{fi}, if the predicate $var_{x,n}$ does not accept trees containing $*$

Conclusion

- An Introduction to Abstract Interpretation
  - Abstract Domain
  - Abstract Transform
  - Galois Connection
- Relational Abstract Interpretation for Verification of 2-hypersafety properties
  - Via self-composition of CFG
  - XML manipulating language

References

[1] P. Cousot and R. Cousot. “Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints”. In 4th POPL, pages 238-252, Los Angeles, CA, 1977. ACM Press.

[2] P. Cousot. MIT Course 16.399: Abstract Interpretation. http://web.mit.edu/afs/athena.mit.edu/course/16/16.399/www/, 2005.

[3] C. Reichenbach, Goethe University Frankfurt Course, Foundations of Programming Languages, http://www.sepl.informatik.uni-frankfurt.de/2014-ws/m-ps/index.en.html, 2014.

[4] M. Kovács, H. Seidl, and B. Finkbeiner. "Relational abstract interpretation for the verification of 2-hypersafety properties." Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM, 2013.

[5] A. Chudnov, G. Kuan, and D. A. Naumann. "Information flow monitoring as abstract interpretation for relational logic." Computer Security Foundations Symposium (CSF), 2014 IEEE 27th. IEEE, 2014.

Any Questions?