CounterExample-Guided Abstraction / Refinement for Symbolic Model Checking

Konkuk University
Department of Computer Science & Engineering
Kunha Kim

Introduction

- In this article, we present an automatic iterative abstraction-refinement methodology that extends symbolic model checking
- Despite the success of symbolic methods, the state explosion problem remains a major hurdle in applying model checking to large industrial designs.
- There are many techniques to reduce the number of the states, such as symmetry reduction, partial order reduction and abstraction techniques.
- Among these techniques, abstraction is considered the most general and flexible for handling the state explosion problem.

Introduction(Con’t)

- There are two kinds of abstraction, **Over-approximation** and **Under-approximation**. In this paper, we’re gonna use Over-approximation.
- They admit only **false negatives** (erroneous counterexamples) and **false positives**, respectively.
- This article describes a new counterexample-guided abstraction technique that extends the general framework of existential abstraction.
- The key step is to extract information from false negatives due to over-approximation.
- Such artificial specification violations are witnessed by what we call “spurious counterexamples”.
- Our method is **complete for an important fragment of ACTL***, precisely for the fragment of ACTL* that admits counterexamples in the form of finite or infinite traces (i.e., finite traces followed by loops).

Recall: CTL and $CTL*$

- State formula ϕ in $CTL*$ is $true \mid p_i \mid \phi_1 \wedge \phi_2 \mid \neg\phi_1 \mid E\ \alpha \mid A\ \alpha$ where $p_i \in AP$, ϕ_1, ϕ_2 are state formula and α is a path formula.
- Path formula α in $CTL*$ is $\phi \mid \neg\alpha_1 \mid \alpha_1 \wedge \alpha_2 \mid X\ \alpha_1 \mid F\ \alpha_1 \mid G\ \alpha_1 \mid \alpha_1 U \alpha_2$ where ϕ is state formula and α_1 and α_2 is a path formula.
- CTL is a subfragment of $CTL*$, where path quantifier A or E is immediately followed by a temporal operator.
- $ACTL*$ is the fragment of $CTL*$, where only the path operator A is used, and negation is restricted to atomic formulas.
- An important feature of $ACTL*$ is the existence of counterexamples.

Simulation and Preservation of $ACTL^*$

- Given two Kripke structures M and M' with $AP \supset AP'$, a relation $H \subseteq S \times S'$ is a $simulation\ relation$ between S and S' if and only if for all $(s, s') \in H$, the following conditions hold.

- $L(s) \cap A' = L'(s')$ and for each state s_1 such that $(s, s_1) \in R$, there exists a state s_1' with the property that $(s', s_1') \in R'$ and $(s_1, s_1') \in H$.

- We say that $M'\ simulates\ M$ (denoted by $M \preccurlyeq M'$) if there exists a simulation relation H such that for every initial state s_0 in M there exists an initial state s_0' in M' for which $(s_0, s_0') \in H$.

- For every $ACTL^*$ formula ρ over atomic proposition AP', if $M \preccurlyeq M'$ and $M' \models \rho$, then $M \models \rho$ (Theorem 2.3).

Abstract function h

- Intuitively speaking, existential abstraction amounts to **partitioning the states** of a Kripke structure **into clusters**, and treating the clusters as new abstract states.

- Formally, an abstraction function h is described by a surjection $h : S \to \hat{S}$ where $\hat{S}$ is the set of abstract states.

- Let d, e be states in S. We say d and e are **logically equivalent on relation** h iff $h(d) = h(e)$ and denoted by $d \equiv_h e$

Formal Definition of Abstraction

- The $abstract\ Kripke\ structure\ \hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$ generated by the abstraction function h is defined as
  - $\hat{I}(\hat{d})$ iff $\exists d(h(d) = \hat{d} \wedge I(d))$
  - $\hat{R}(\hat{d}_1, \hat{d}_2)$ iff $\exists d_1 \exists d_2(h(d_1) = \hat{d}_1 \wedge h(d_2) = \hat{d}_2 \wedge R(d_1, d_2))$
  - $\hat{L}(\hat{d}) = \cup_{h(d)=\hat{d}} L(d)$
- An abstraction function h is $appropriate$ for a specification ρ if for all atomic sub-formulas f of ρ and for all states d and e in the domain S such that $d \equiv_h e$ it holds $d \vDash f \Leftrightarrow e \vDash f$.

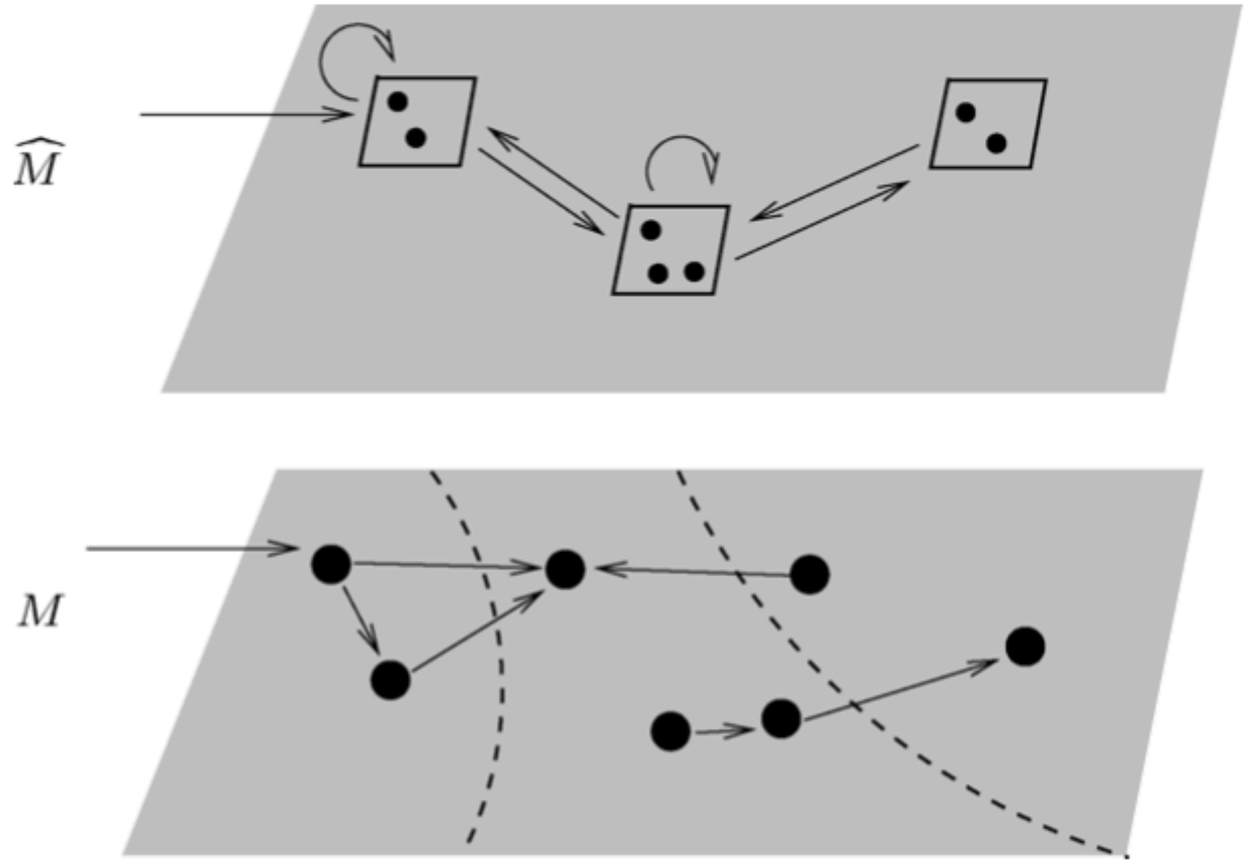


FIG. 1. Existential Abstraction. M is the original Kripke structure, and $\widehat{M}$ the abstracted one. The dotted lines in M indicate how the states of M are clustered into abstract states.

Formal Definition of Abstraction(Con't)

- h is appropriate for a set F of formulas if h is appropriate for all $f \in F$.
- $\hat{L}(\hat{d})$ is $consistent$ if all concrete states corresponding to $\hat{d}$ satisfy all labels in $\hat{L}(\hat{d})$. That is, collapsing a set of concrete states into an abstract state does not lead to contradictory labels.
- If h is appropriate for ρ, then all concrete states in an equivalence class of $\equiv_h$ share the same labels.
- Also, the abstract states inherit all labels from each state in the respective equivalence classes
- Finally, the labels of the abstract states are consistent.
- In other words, $d \equiv_h d'$ implies $L(d) = L(d')$, $h(d) = \hat{d}$ implies $\hat{L}_h(\hat{d}) = L(d)$, and $\hat{L}_h(d)$ is consistent.

Formal Definition of Abstraction(Con't)

- Let h be appropriate for the $ACTL^*$ specification ρ and M be defined over the atomic propositions in ρ. Then $M \preccurlyeq \hat{M}_h$ (Lemma 3.2).
- A simulation relation $H = \{(s, h(s) \mid s \in S\}$ trivially satisfies Lemma 3.2.
- Let h be appropriate for the $ACTL^*$ specification ρ. Then $\hat{M}_h \models \rho \ implies \ M \models \rho$
-

Example: Traffic light

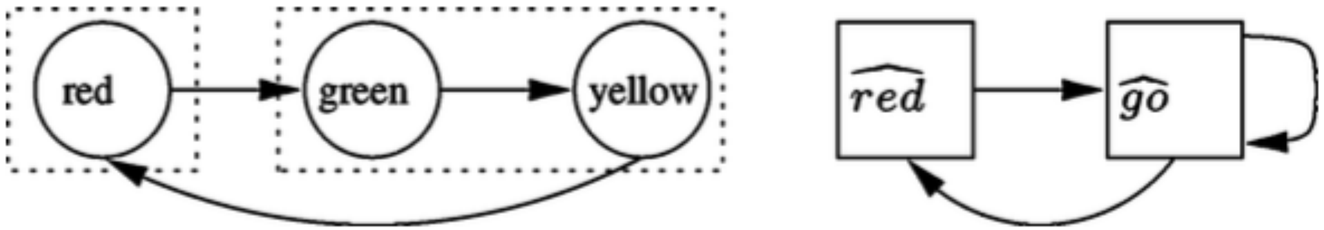


FIG. 2. Abstraction of a US traffic light.

- We want to prove $\psi = AG(AF(state = red))$ using the abstraction function $h(red) = \hat{red}$ and $h(green) = h(blue) = \hat{go}$
- It is easy to see that $M \models \psi$ while $\hat{M} \not\models \psi$. There exists an infinite abstract trace $< \hat{red}, \hat{go}, \hat{go}, \cdots >$ that invalidates ψ. This is a spurious counterexample.

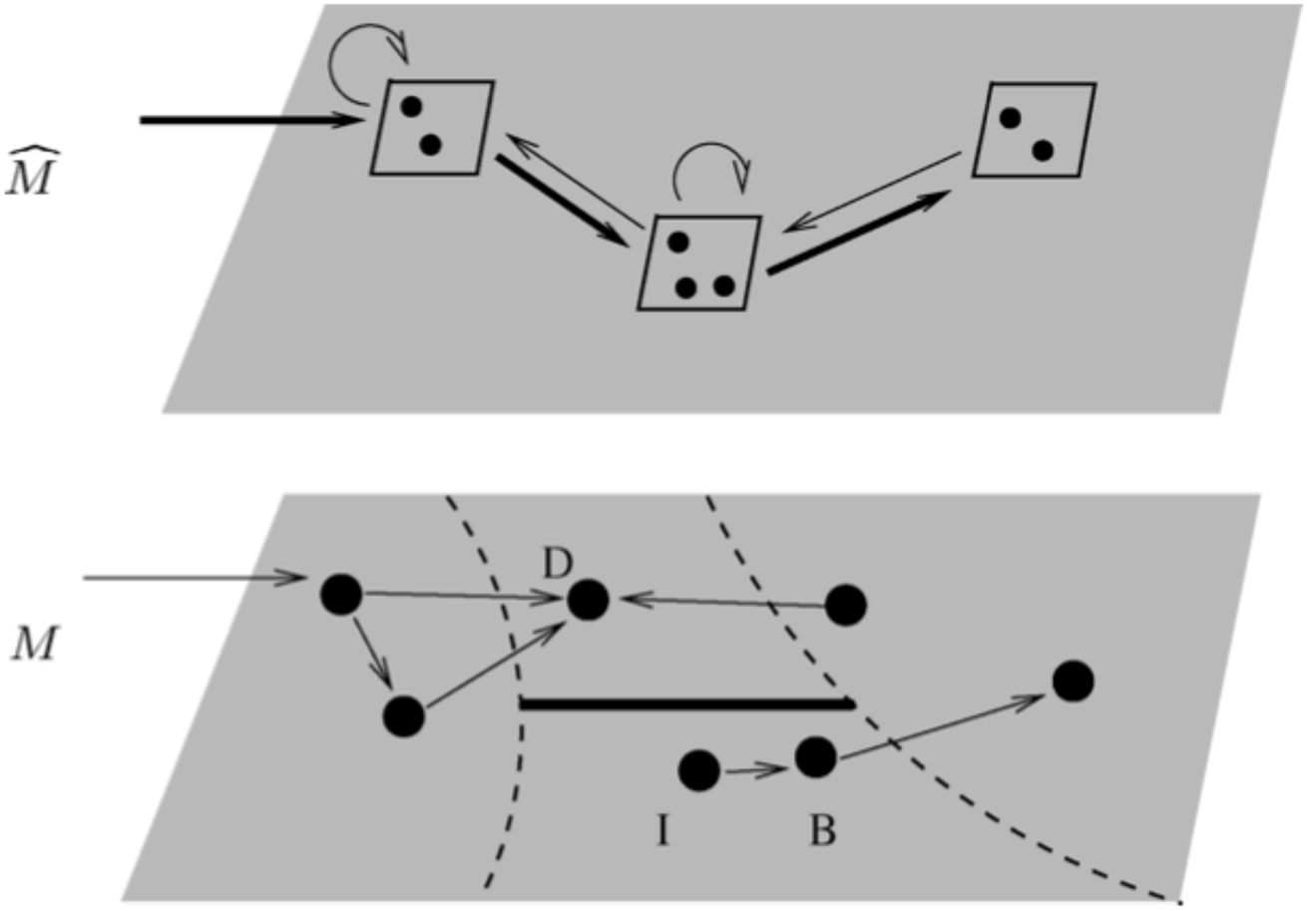


FIG. 3. The abstract path in $\widehat{M}$ (indicated by the thick arrows) is spurious. To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line in M.

Clue of Refinement

- We see that the abstract path does not have a corresponding concrete path.
- Whichever concrete path we follow, we will end up in state D, from which we cannot go any further. Therefore, D is called a deadend state.
- On the other hand, the bad state is state B, because it made us believe that there is an outgoing transition.
- In addition, there is the set of irrelevant states I that are neither deadend nor bad, and it is immaterial whether states in I are combined with D or B.
- To eliminate the spurious path, the abstraction has to be refined as indicated by the thick line.

Approximation for Existential Abstraction

- It is too expensive to compute existential abstraction directly. Instead, we use approximation.
- If a Kripke structure $\tilde{M} = (S_h, \tilde{I}, \tilde{R}, \hat{L}_h)$ satisfies $\hat{I}_h \subseteq \tilde{I}$ and $\hat{R}_h \subseteq \tilde{R}$, we say $\tilde{M}$ *approximates* $\hat{M}_h$.
- Let h be the appropriate for the $ACTL^*$ specification ρ. Then $\hat{M}_h \preccurlyeq \tilde{M}$ (Theorem 3.5)
- A simulation relation is given by the identity mapping that maps each state in M to the same state in $\hat{M}_h$ – note that the sets of states in M and $\hat{M}_h$ coincide.

"*Early*" Approximation

- Since $\preccurlyeq$ is preorder, $M \preccurlyeq \hat{M}_h \preccurlyeq \tilde{M}$ in accordance with Theorem 3.3 and 3.5 define a practical transformation T called $early\ approximation$.
- T applies existential abstraction operation directly to variables at the innermost level of the formula.
- This transformation generates a new structure $\tilde{M}_t = (S_h, T(I), T(R), \hat{L}_h)$ where $T(R) = (R_1)_h \wedge \cdots \wedge (R_n)_h$ and analogously for I $(R = R_1 \wedge \cdots \wedge R_n)$.
- R_i defines the transition relation for single variable.

"*Early*" Approximation(Con't)

- Consider a system M, which is a synchronous composition of M_1 and M_2. ($M = M_1 || M_2$)
- Both M_1 and M_2 define the transition of one variable. In this case, $\tilde{M}_T$ is equal to $(\hat{M}_1)_h || (\hat{M}_2)_h$.(Abstraction applied for each model)
- Since T is applied at the innermost level, abstraction can be performed before building the BDDs for the transition relation.
- It is easy to implement and fast. But it has potential limitations in checking certain properties.
- Since $\tilde{M}_T$ is a coarse abstraction, there exist many properties that cannot be checked on $\tilde{M}_T$, but can still be verified using a finer approximation.

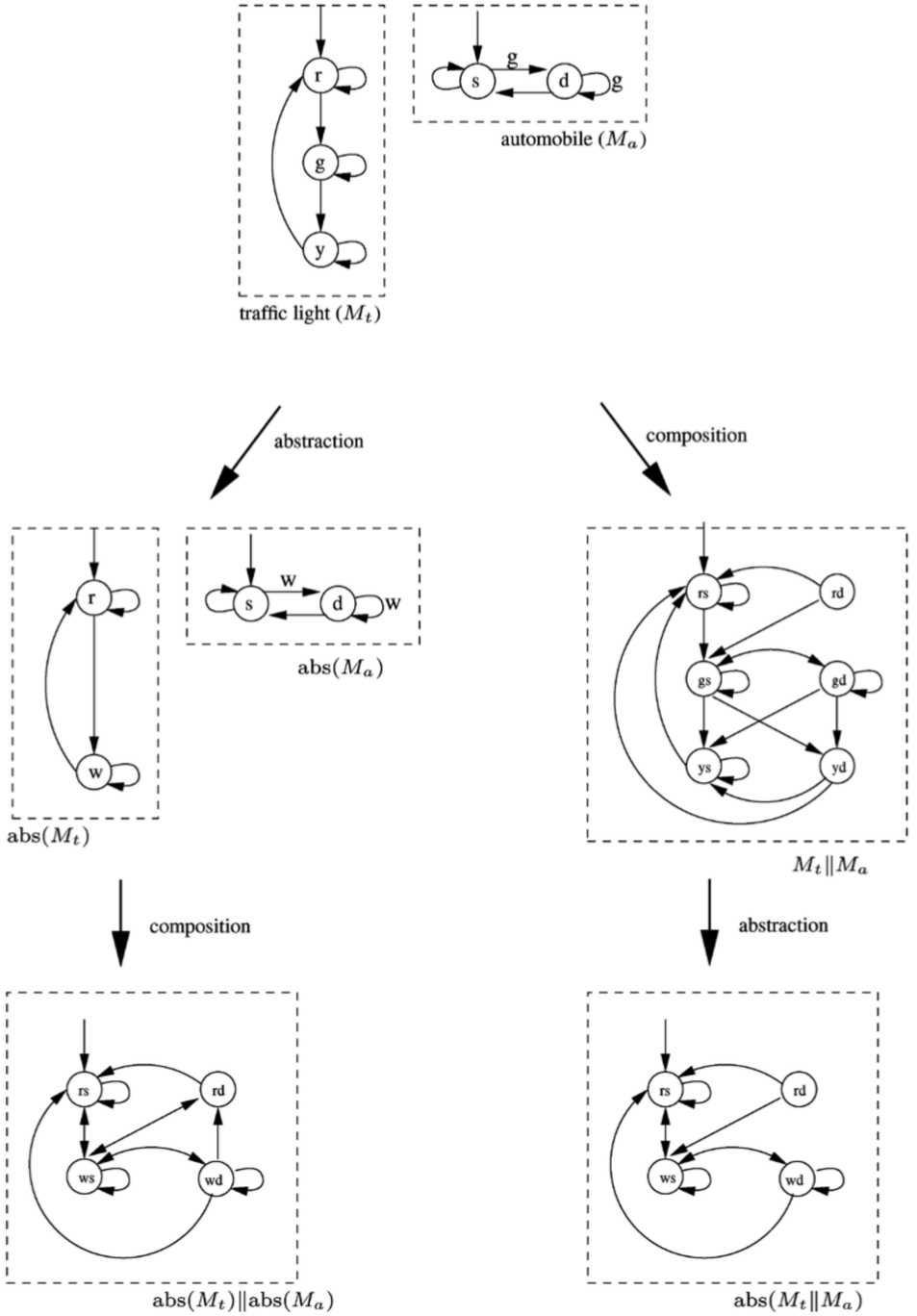


FIG. 4. Traffic light example.

$$
\begin{array}{l}
\textbf{init}(t) := r; \\
\textbf{next}(t) := \textbf{case} \\
\quad t = r : \{r, g\}; \\
\quad t = g : \{g, y\}; \\
\quad t = y : \{y, r\}; \\
\textbf{esac;}
\end{array}
$$

$$
\begin{array}{l}
\textbf{init}(c) := s; \\
\textbf{next}(c) := \textbf{case} \\
\quad t = g : \{s, d\}; \\
\quad t \neq g : s; \\
\textbf{esac;}
\end{array}
$$

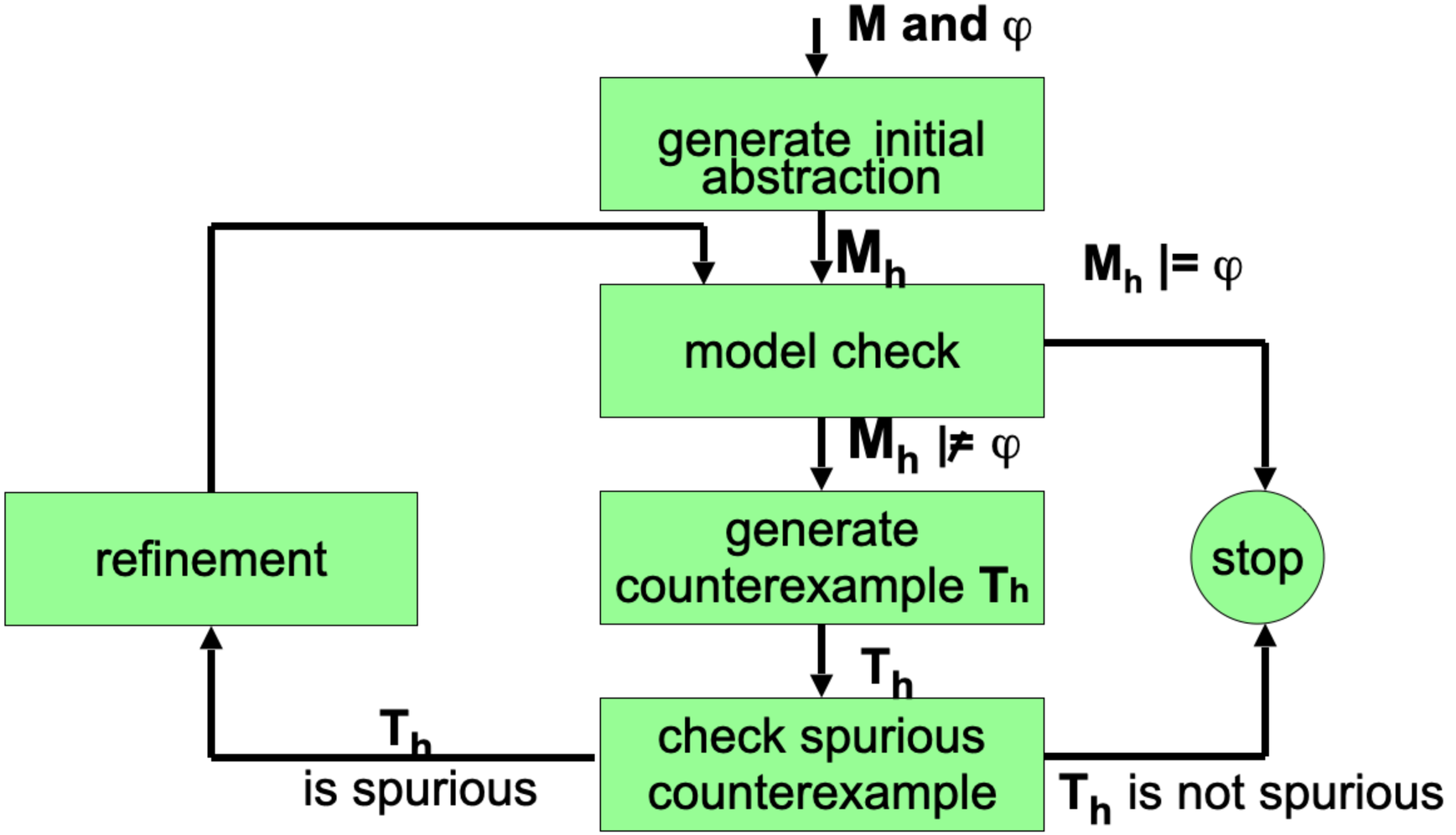
Example: Traffic light and Car

- The Kripke Structure for the system is obtained by composing M_t and M_c.
- The safety property we want to prove is $\rho = AG[\neg(t = r \wedge c = d)]$
- Let us consider an abstraction function abs which maps the value g and d to w. If we apply abs **after composition**, ρ still holds.
- If however, we use the transformation T, which applies abs **before** composition, ρ does not holds.
- We see that when we first abstract the individual components and then compose we may introduce too many spurious behaviors.

A brief view

- It is desirable to obtain an approximation structure $\tilde{M}$ which is more precise than the structure $\tilde{M}_T$ obtained by the technique proposed in Clarke et al.[1994].
- All the transitions in the abstract structure $\hat{M}_h$ are included in both $\tilde{M}$ and $\tilde{M}_T$. Note that the state sets of $\hat{M}_h$, $\tilde{M}$, $\tilde{M}_T$ are the same and $M \preccurlyeq \hat{M}_h \preccurlyeq \tilde{M}_T$.
- $\hat{M}_h$ is intended to be built but too expensive, $\tilde{M}_T$ is easy to build but extra behaviors are added into the structure.
- Our aim is to build a model $\tilde{M}$ which is computationally easier but a more refined approximation of $\hat{M}_h$ than $\tilde{M}_T$.
- It means, $M \preccurlyeq \hat{M}_h \preccurlyeq \tilde{M} \preccurlyeq \tilde{M}_T$.

Our Abstraction Methodology (CAV'2000)



Generating Initial Abstraction(Specifically)

- Assume that we have a given program P with n variables $\{v_1, v_2, \cdots, v_n\}$.
- Given an atomic formula f, let $var(f)$ is set of variables appearing in f.
  For example, $var(x = y)$ is $\{x, y\}$.
- Given a set of atomic formulas U, $var(U)$ equals $\cup_{f \in U} var(f)$.
- Let $\equiv_I$ be the equivalence relation on $Atoms(P)$, which is reflexive and transitive. The equivalence class of an atomic formula $f \in Atoms(P)$ is called $formula\ cluster$ of f, denoted by $[f]$.
- Let f_1 and f_2 be an atomic formulas. $var(f_1) \cap var(f_2) \neq \emptyset$ means $[f_1] = [f_2]$.
- It means a variable v_i cannot appear in formulas that belong to two different formula clusters.

Generating Initial Abstraction(Cont.)

- The formula clusters induce an equivalence relation $\equiv_V$ on the set of variables V by the following manner :
- $v_i \equiv_V v_j$ iff v_i and v_j appear in atomic formulas that belong to the same formula cluster.
- The equivalence class of $\equiv_V$ are called $variable\ clusters$. For instance, consider $FC_i = \{v_1 > 3, v_1 = v_2\}$. The corresponding variable cluster is $VC_i = \{v_1, v_2\}$.
- Now, let $\{FC_1, FC_2, \cdots, FC_m\}$ be the set of formula clusters and $\{VC_1, VC_2, \cdots, VC_m\}$ the set of corresponding variable clusters.
- We construct initial abstraction $h = (h_1, h_2, \cdots, h_m)$ as follows.
- For each h_i, we set $D_{VC_i} = \Pi_{v \in VC_i} D_v$, i.e. D_{VC_i} is the domain corresponding to the variable cluster VC_i. The corresponding abstraction h_i is defined on D_{VC_i} as follows : $h_i(d_1, \cdots, d_k) = h_i(e_1, \cdots, e_k)$ iff $(d_1, \cdots, d_k) \models f \Leftrightarrow (e_1, \cdots, e_k) \models f$.

Identification of Spurious Counterexamples: Path

- First we will tackle the case $\hat{T}$ is a path $< \hat{s}_1, \cdots, \hat{s}_n >$. Given an abstract state $\hat{s}$, the set of concrete states s s.t. $h(s) = \hat{s}$ is denoted by $h^{-1}(\hat{s})$.

- We extend h^{-1} to sequences in the following way: $h^{-1}(\hat{T})$ is the set of concrete paths given by the following expression:
  $\{ < s_1, \cdots, s_n > \mid \wedge_{i=1}^{n} h(s_i) = \hat{s}_i \ \wedge I(s_1) \ \wedge \ \wedge_{i=1}^{n-1} R(s_i, s_{i+1}) \}$

- Let $S_1 = h^{-1}(\hat{s}_1) \cap I$. For $1 < i \leq n$, we define S_i in the following manner:
  $S_i \doteqdot Img(S_{i-1}, R) \cap h^{-1}(\hat{s}_i)$. This can be computed by using OBDD and standard image computation algorithm.

- The following lemma establishes the correctness of this procedure.

- $Lemma\ 1$. If the path $\hat{T}$ corresponds to a concrete counterexample, the set of concrete paths $h^{-1}(\hat{T})$ is non-empty and $S_i \neq \emptyset$ for all $1 \leq i \leq n$.

Algorithm SplitPATH

$S := h^{-1}(\widehat{s_1}) \cap I$

$j := 1$

while $(S \neq \emptyset \text{ and } j < n)$ {

$j := j + 1$

$S_{\text{prev}} := S$

$S := Img(S, R) \cap h^{-1}(\widehat{s_j})$ }

if $S \neq \emptyset$ **then** output counterexample

else output j, S_{prev}

Fig. 4. SplitPATH checks spurious path.

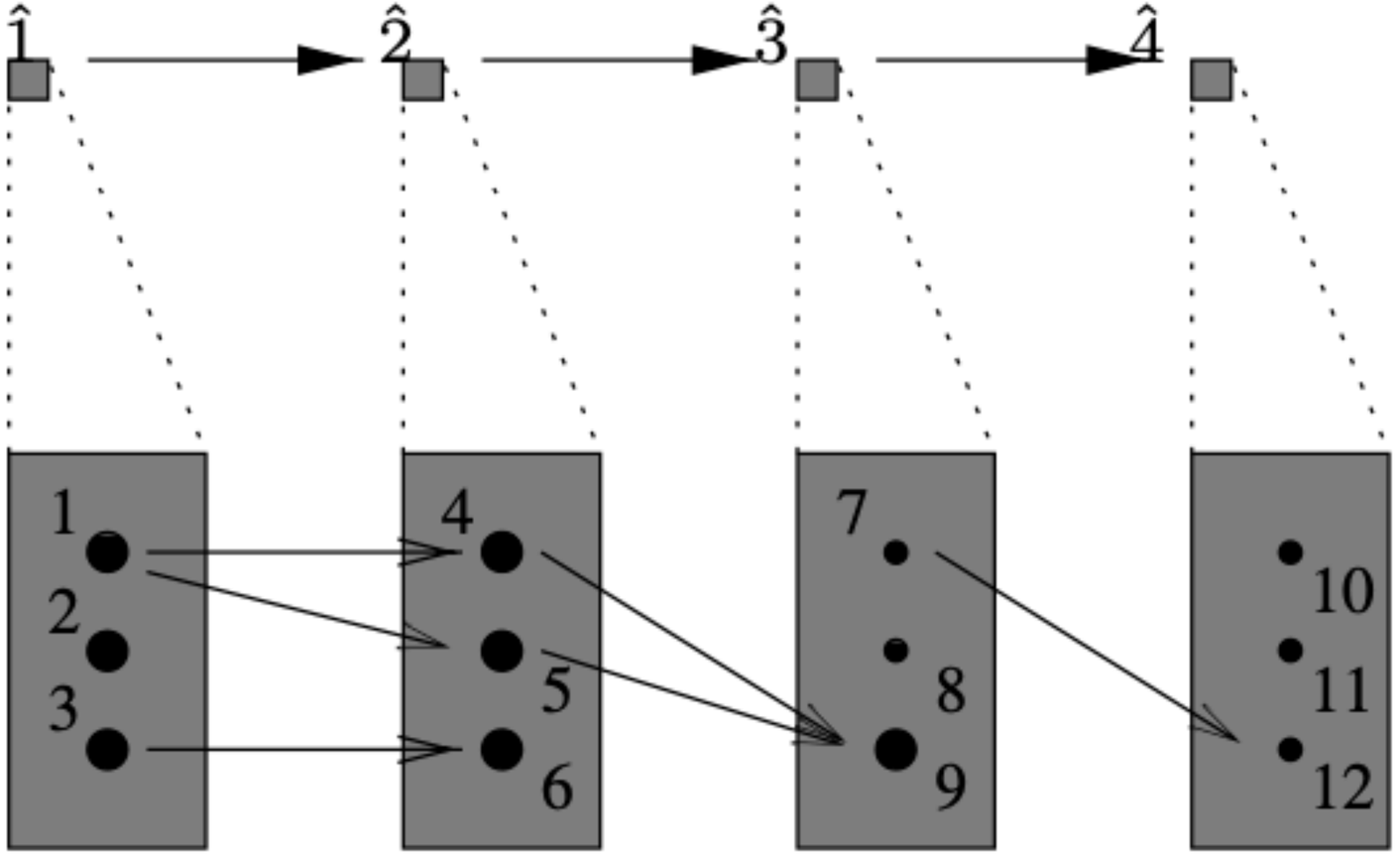


Fig. 3. An abstract counterexample

Example

- Consider a program with only one variable with domain $D = \{1,\cdots,12\}$. Assume that h maps $x \in D$ to $\lfloor (x-1)/3 \rfloor + 1$.
- There are four abstract states $\hat{1}, \hat{2}, \hat{3}$ and $\hat{4}$ correspond to $\{1,2,3\}, \{4,5,6\}, \{7,8,9\}, \{10,11,12\}$ respectively.
- Suppose we obtain an abstract counterexample $\hat{T} = < \hat{1}, \hat{2}, \hat{3}, \hat{4} >$. It is easy to see $\hat{T}$ is spurious.
- Also, we can check $S_1 = \{1,2,3\}, S_2 = \{4,5,6\}, S_3 = \{9\}, S_4 = \emptyset$. Notice that S_4 and therefore $Img(S_3, R)$ are both empty.
- The algorithm **SplitPATH** will output 3 and S_3.

Identification of Spurious Counterexamples: Loop

- Now we're gonna focus on loop counterexample, described as $< \hat{s}_1, \cdots \hat{s}_i > < \hat{s_{i+1}}, \cdots \hat{s}_n >^{\omega}$. The loop starts at $\hat{s_{i+1}}$ and ends at $\hat{s}_n$.

- Consider a loop $< \hat{s}_1 > < \hat{s}_2, \hat{s}_3 >^{\omega}$. To detect this loop is corresponds to concrete loop or not, we unwind the counterexample.

- As you can see in the example, a given abstract loop may correspond to several concrete loops of different size(in this case, 3 and 6)

- In addition, each individual loops may start at different stages of unwinding. Also, it eventually becomes periodic ($S_3^0 = S_3^2$).

- The size of the period is the least common multiple of the size of the individual loops, and thus, in general **exponential**.

Identification of Spurious Counterexamples: Loop(Cont.)

- However, for $\hat{T} = < \hat{s}_1, \cdots \hat{s}_i > < \hat{s_{i+1}}, \cdots \hat{s}_n >^{\omega}$, the number of unwinding can be bounded by $min = \min(|h^{-1}(\hat{s}_j)|)(i+1 \leq j \leq n)$.
- In other words, the number of unwinding is at most the number of concrete states for any abstract state in the loop.
- Let $\hat{T}_{unwind}$ denote $< \hat{s}_1, \cdots \hat{s}_i > < \hat{s_{i+1}}, \cdots \hat{s}_n >^{min+1}$. If $\hat{T}$ corresponds to concrete counterexample, $h^{-1}(\hat{T}_{unwind})$ is not empty.
- This algorithm is more complicated than the path counterexamples.

Algorithm SplitLOOP($\widehat{T}$)

$min = \min\{|h^{-1}(\widehat{s_{i+1}})|, \ldots, |h^{-1}(\widehat{s_n})|\}$
$\widehat{T}_{\text{unwind}} = \textbf{unwind}(\widehat{T}, min + 1)$
Compute j and S_{prev} as in **SplitPATH**($\widehat{T}_{\text{unwind}}$)
$k := \textbf{LoopIndex}(j)$
$p := \textbf{LoopIndex}(j + 1)$
output S_{prev}, k, p

FIG. 9. SplitLOOP checks if an abstract loop is spurious.

Algorithm SplitLOOP

- This algorithm is an extension of SplitPATH. $\hat{T}_{unwind}$ is computed by the subprogram **unwind.**
- The subprogram LoopIndex(j) computes the index of the abstract state at position j in the unwound counterexample $\hat{T}_{unwind}$.
- That is, for $j \leq n$, LoopIndex outputs j, and for $j > n$, LoopIndex outputs $[j - (i+1)\ mod(n-i)] + (i+1)$
- If the abstract counterexample is spurious, then the algorithm SplitLOOP outputs a set S_{prev} and indices k, p, such that the following conditions hold.

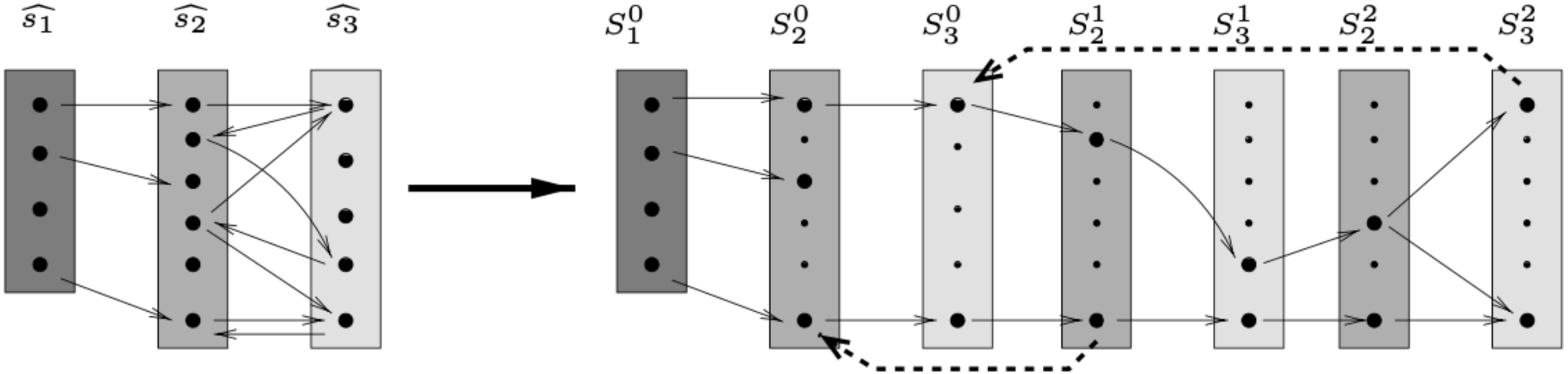


Fig. 5. A loop counterexample, and its unwinding.

(1) The states in S_{prev} correspond to the abstract state $\widehat{s_p}$, that is, $S_{\text{prev}} \subseteq h^{-1}(\widehat{s_p})$.
(2) All states in S_{prev} are reachable from $h^{-1}(\widehat{s_1}) \cap I$.
(3) k is the successor index of p within the loop, that is, if $p = n$ then $k = i + 1$, and otherwise $k = p + 1$.
(4) There is no transition from a state in S_{prev} to $h^{-1}(\widehat{s_k})$, that is, $Img(S_{\text{prev}}, R) \cap h^{-1}(\widehat{s_k})$ is empty.
(5) Therefore, $\widehat{s_p}$ is the failure state of the loop counterexample.

Refining the Abstraction

- First, we will consider counterexample $\hat{T} = < \hat{s}_1, \cdots, \hat{s}_n >$ is a path.
- Since $\hat{T}$ does not correspond to a real counterexample, there exists a set $S_i \subseteq h^{-1}(\hat{s}_i)$ with $1 \leq i < n$ such that $Img(S_i, R) \cap h^{-1}(\hat{s}_{i+1}) = \varnothing$ and S_i is reachable from initial state set $h^{-1}(\hat{s}_1) \cap I$.
- Since there is a transition from $\hat{s}_i$ to $\hat{s}_{i+1}$, there is at least one transition from a state in $h^{-1}(\hat{s}_i)$ to $h^{-1}(\hat{s}_{i+1})$ even though there is no transition from S_i to $h^{-1}(\hat{s}_{i+1})$.
- Our goal is to separate the dead-end state and bad state. To do this, we partition $h^{-1}(\hat{s}_i)$ into three pieces.
- $S_{i,0} = S_i,\ S_{i,1} = \{s \in h^{-1}(\hat{s}_i) \mid \exists s' \in h^{-1}(\hat{s}_{i+1}) \,.\, R(s, s')\}, S_{i,x} = h^{-1}(\hat{s}_i) \backslash (S_{i,0} \cup S_{i,1})$
- Each of these subsets are correspond to dead-end state, bad state, irrelevant state, respectively.

Recall : Fig3

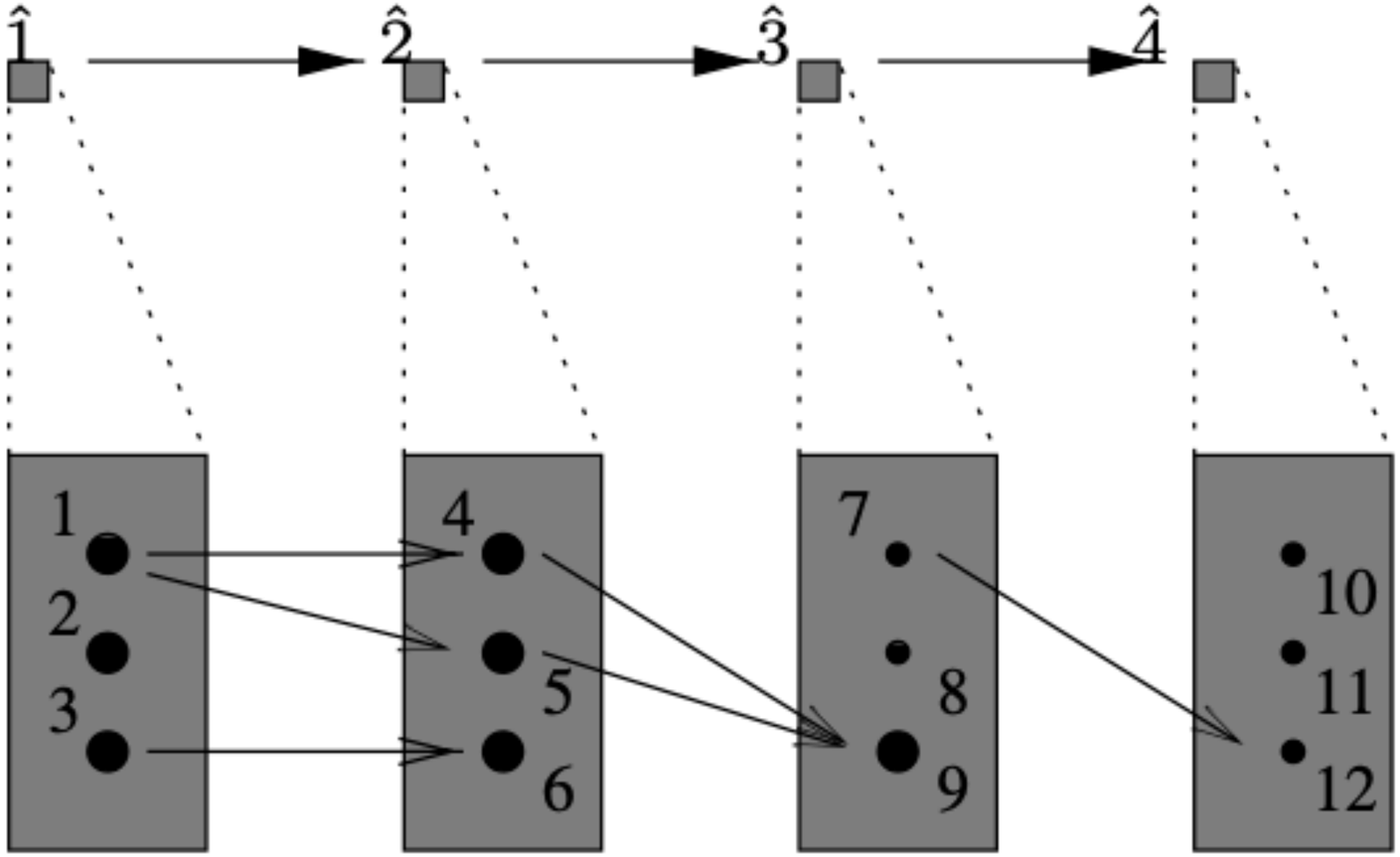


Fig. 3. An abstract counterexample

Example: Figure 3

- We can easily obtain $S_1 = \{1,2,3\}, S_2 = \{4,5,6\}, S_3 = \{9\}, S_4 = \emptyset$.
- Also, using our definition, we have $S_{3,0} = \{9\}, S_{3,1} = \{7\}, S_{3,x} = \{8\}$.
- Since $S_{i,1}$ is not empty, there is a spurious transition $\hat{s}_i \rightarrow \hat{s}_{i+1}$.
- As we mentioned, we need to separate $S_{i,0}$ and $S_{i,1}$. We need an abstract function, in which no abstract state simultaneously contains states from $S_{i,0}$ and $S_{i,1}$.

Refining the Abstraction(Cont.)

- It is natural to describe the needed refinement in terms of equivalence relations. Recall that $h^{-1}(\hat{s})$ is an equivalence class of $\equiv$ which has the form $E_1 \times \cdots \times E_m$, where E_i is an equivalence class of $\equiv_i$.

- Thus, the refinement is obtained by partitioning the equivalence classes E_j into subclasses, which amounts to refining the equivalence relations $\equiv_j$.

- The size of refinement is the number of new equivalence classes. Ideally, we would like to find the coarsest refinement, but this problem in NP-hard.

- But if $S_{i,x} = \emptyset$, the problem can be solved in polynomial time.

- Now, we're gonna focus on this case.

Algorithm PolyRefine

- Let P_j^+, P_j^- be two projection functions, such that for $s = (d_1, \cdots, d_m)$, $P_j^+(s) = d_j$ and $P_j^-(s) = (d_1, \cdots, d_{j-1}, d_{j+1}, \cdots, d_m)$.
- A $proj(S_{i,0}, j, a)$ denotes the projection set $\{P_j^- \mid P_j^+(s) = a, s \in S_{i,0}\}$.
- Intuitively, $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$ means that there exists $(d_1, \cdots, d_{j-1}, d_{j+1}, \cdots, d_m) \in proj(S_{i,0}, j, a)$ and $(d_1, \cdots, d_{j-1}, d_{j+1}, \cdots, d_m) \notin proj(S_{i,0}, j, b)$
- By the definition of $proj(S_{i,0}, j, a)$, $s_1 = (d_1, \cdots, d_{j-1}, a, d_{j+1}, \cdots, d_m) \in S_{i,0}$ and $s_2 = (d_1, \cdots, d_{j-1}, a, d_{j+1}, \cdots, d_m) \notin S_{i,0}$. Therefore, $s_2 \in S_{i,1}$.
- So, we need to change $\equiv_j$ to $\equiv_j'$ such that $a \not\equiv_j' b$. Also, $\equiv_j'$ is the coarsest refinement of $\equiv_j$.
- In practical, we merge the states in $S_{i,x}$ to $S_{i,1}$ and use PolyRefine.

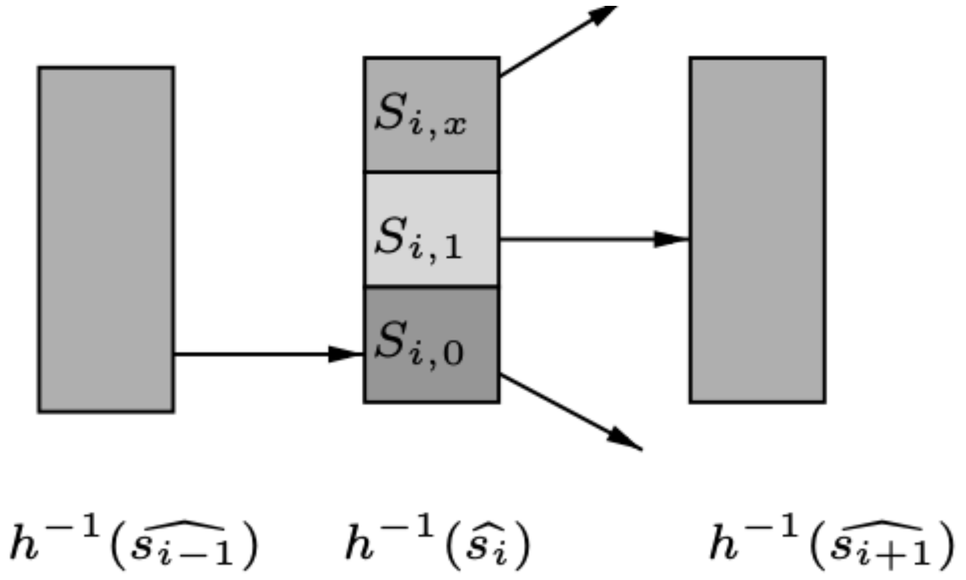


Fig. 6. Three sets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$

Algorithm PolyRefine

for j := 1 **to** m {
  $\equiv'_j := \equiv_j$
  for every $a, b \in E_j$ {
    if $proj(S_{i,0}, j, a) \neq proj(S_{i,0}, j, b)$
      then $\equiv'_j := \equiv'_j \setminus \{(a, b)\}$ }}

Fig. 7. The algorithm **PolyRefine**

Proof of PolyRefine

LEMMA 4.19. *When $\mathcal{S}_{\mathrm{I}} = \emptyset$, the relation $\equiv'$ computed by* **PolyRefine** *is an equivalence relation which refines $\equiv$ and separates $\mathcal{S}_{\mathrm{D}}$ and $\mathcal{S}_{\mathrm{B}}$.*

PROOF. The algorithm **PolyRefine** computes

$$\begin{aligned}
\equiv'_j &= \equiv_j - \{(a,b) : \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) \neq \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b) \\
&= \{(a,b) : a \equiv_j b \wedge \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, a) = \mathrm{proj}(\mathcal{S}_{\mathrm{D}}, j, b)\} \\
&= \{(a,b) : a \equiv_j b \wedge \forall d_1, \ldots, d_m. \\
&\qquad (d_1, \ldots, d_{j-1}, a, d_{j+1}, \ldots, d_m) \in \mathcal{S}_{\mathrm{D}} \leftrightarrow \\
&\qquad (d_1, \ldots, d_{j-1}, b, d_{j+1}, \ldots, d_m) \in \mathcal{S}_{\mathrm{D}}\}
\end{aligned}$$

- Note that we view equivalence relations as sets of pairs of states, as subsets of $S \times S$.

Proof of PolyRefine(Cont.)

- Now, we show that $\equiv'$ is a correct refinement, that is, for all dead-end states $d \in S_D$ and bad states $b \in S_B$ it holds that $d \not\equiv' b$.

- Let b and d be such states, and assume towards a contradiction, that $b \equiv' d$. Recall that $h(b) = h(d) = \hat{s}_i$ and $b \equiv d$ ($\hat{s}_i$ is a failure state).

- By construction, b and d have the form $b = (b_1, \cdots, b_m)$, $d = (d_1, \cdots, d_m)$ where for all $i \in [1,m]$, b_i and d_i are in D_i.

- Consider a sequence $x_1, \cdots, x_{m+1}$ of states constructed as follows:
  $x_1 = (b_1, b_2, \cdots, b_m) = b, x_2 = (d_1, b_2, \cdots, b_m), \cdots,$
  $x_m = (d_1, \cdots, d_{m-1}, b_m), x_{m+1} = (d_1, \cdots, d_m) = d$

- All x_i are concrete states corresponding to the failure state. It means,
  $h(x_j) = (h(b_1), \cdots, h(b_m)) = \hat{s}_i$

State Splitting

- For a set $h^{-1}(\hat{s}_i)$, a set of concrete states correspond to $\hat{s}_i$ occurring in spurious counterexample, split it into $h^{-1}(\hat{s}_i) \cap Image(h^{-1}(\hat{s}_{i-1}))$ and $h^{-1}(\hat{s}_i) \backslash Image(h^{-1}(\hat{s}_{i-1}))$, provided both non-empty.
- In the case $i = 1$, split it into $h^{-1}(\hat{s}_1) \cap I$ and $h^{-1}(\hat{s}_1) \backslash I$.
- In other words, replace $\hat{s}_i$ by two states $\hat{s}_i^+$ and $\hat{s}_i^-$ representing $h^{-1}(\hat{s}_i) \cap Image(h^{-1}(\hat{s}_{i-1}))$ and $h^{-1}(\hat{s}_i) \backslash Image(h^{-1}(\hat{s}_{i-1}))$, respectively.
- Therefore, $\hat{S}$ is turned into $\hat{S}' = \hat{S} \backslash \{\hat{s}_i\} \cup \{\hat{s}_i^+, \hat{s}_i^-\}$.

State Splitting(Cont.)

- So, replace $\hat{M} = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$ to $\hat{M}' = (\hat{S}', \hat{I}', \hat{R}', \hat{L}')$ denoted as

$$\hat{S}' = \hat{S} \backslash \{\hat{s}_i\} \cup \{\hat{s}_i^+, \hat{s}_i^-\}$$

$$\hat{I}' = \hat{I} \text{ if } h^{-1}(\hat{s}_i) \cap I = \varnothing, \text{ otherwise } I \backslash \{\hat{s}_i\} \cup \{\hat{s}_i^+\}$$

$$\hat{R}' = \hat{R} \cap (\hat{V}' \times \hat{V}') \cup \{(\hat{s}_i^+, \hat{s}_i^-), (\hat{s}_i^-, \hat{s}_i^+)\} \cup \{(\hat{s}, \hat{s}_i^+) \,|\, (\hat{s}, \hat{s}_i) \in \hat{R}\} \cup \{(\hat{s}, \hat{s}_i^-) \,|\, (\hat{s}, \hat{s}_i) \in \hat{R}, \hat{s} \neq \hat{s}_{i-1}\} \cup \{(\hat{s}_i^+, \hat{s}), (\hat{s}_i^-, \hat{s}) \,|\, (\hat{s}_i, \hat{s}) \in \hat{R}\}$$

$$\hat{L}'(\hat{s}) = L(\hat{s}) \text{ if } \hat{s} \in \hat{S}, L(\hat{s}_i) \text{ if } \hat{s} \in \{\hat{s}_i^+, \hat{s}_i^-\}$$

Abstraction and Refining transition

- So, resulting abstraction is $h'(s) = \hat{s}_i^+$ if $s \in h^{-1}(\hat{s}_i) \cap Image(\hat{s}_{i-1})$, $\hat{s}_i^-$ if $s \in h^{-1}(\hat{s}_i) \backslash Image(\hat{s}_{i-1})$, otherwise $h(s)$.

- Pre- and post-images of $h'^{-1}(\hat{s}_i^+)$ or $h'^{-1}(\hat{s}_i^-)$ may well have empty intersections with sets that the pre- or post-set of $h'^{-1}(\hat{s}_i)$ did intersect with. In such cases, $\hat{R}'$ contains spurious edges.

- Thus, remove such edges by pruning $\hat{R}'$ to $\hat{E}'' = \{(s,t) \in \hat{R}' \mid Image(h'^{-1}(s)) \cap h'^{-1}(t) \neq \emptyset\}$

Our Abstraction Methodology (CAV'2000)