UiO : Department of Informatics
University of Oslo

Counterexample Guided Abstraction Refinement (CEGAR)

Peyman Rasouli
Gianluca Turin

Counterexample-guided Abstraction Refinement *

Edmund Clarke[1], Orna Grumberg[2], Somesh Jha[1], Yuan Lu[1], and Helmut Veith[1,3]

[1] Carnegie Mellon University, Pittsburgh, USA [2] Technion, Haifa, Israel
[3] Vienna University of Technology, Austria

Abstract. We present an automatic iterative abstraction-refinement methodology in which the initial abstract model is generated by an automatic analysis of the control structures in the program to be verified. Abstract models may admit erroneous (or "spurious") counterexamples. We devise new symbolic techniques which analyze such counterexamples and refine the abstract model correspondingly. The refinement algorithm keeps the size of the abstract state space small due to the use of abstraction functions which distinguish many degrees of abstraction for each program variable. We describe an implementation of our methodology in NuSMV. Practical experiments including a large Fujitsu IP core design with about 500 latches and 10000 lines of SMV code confirm the effectiveness of our approach.

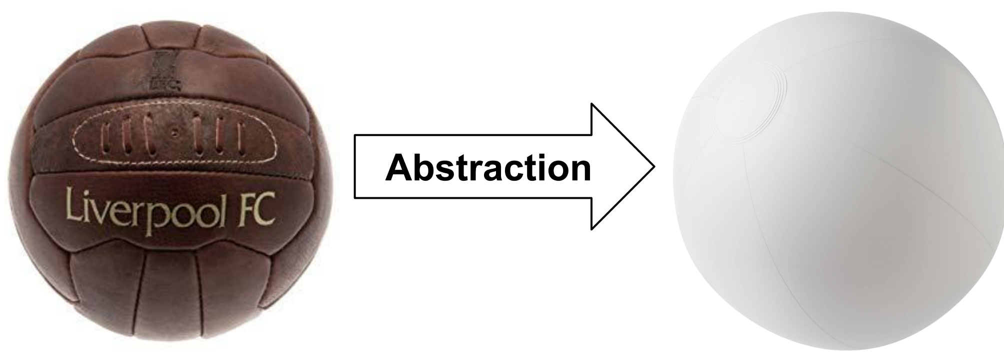
1 Introduction

The state explosion problem remains a major hurdle in applying model checking to large industrial designs. Abstraction is certainly the most important technique for handling this problem. In fact, it is essential for verifying designs of industrial complexity. Currently, abstraction is typically a manual process, often requiring considerable creativity. In order for model checking to be used more widely in industry, automatic

Abstraction

- Definition of abstraction on Wikipedia:
  - **Abstractions** may be formed by reducing the information content of a concept or an observable phenomenon, typically to retain only information which is relevant for a particular purpose.

- Example:



Concrete Object Abstract Object

Abstraction in Model Checking

- **State Explosion Problem:** the size of the system state space grows *exponentially* with the number of state variables in the system.
- Translating system/program into a Kripke structure $M = (S, I, R, L)$
  - **Challenge:** constructing and saving a **naive** Kripke structure on a computer is impossible due to its **size** (state explosion problem)
- Obtaining an abstraction of the created structure $\widehat{M} = (\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L})$
- **Abstraction** is aimed at **reducing the state space** for the system by **omitting irrelevant details** to the property being verified.

How abstraction helps?

It assumes reduction in the information content

⇓

results in a reduction of the size of the Kripke structure M

⇓

irrelevant information to the valuation of temporal properties is omitted

⇓

In the end, abstraction not need to be a Kripke structure, but it should allow evaluating temporal properties on that

Semantic Interpretation

- Notion of abstraction:
  - defined via a function ***h*** mapping a Kripke structure to its abstraction.

- **However,** constructing a concrete Kripke structure and then applying ***h*** to it is often impossible
  - *due to potentially too big or even infinite Kripke structure*

- **Therefore,** abstractions are built by applying **“non-standard” semantic interpretations** to system descriptions

Refinement Concept

A specification **True** in the abstract model

⇓

It will also be **True** in the concrete design

A specification **False** in the abstract model, generate **counterexample**

⇓

The **counterexample** may be the result of some behavior in the abstract model which is not present in the concrete design

⇓

Refine the abstraction so that the behavior caused the **erroneous counterexample** is eliminated

Counterexample Guided Abstraction Refinement (CEGAR) Clarke et al., 2000

- Automatic refinement technique for **ACTL* properties.**
- Based on analysis of the structure of formulas appearing in the program.
- Uses information obtained from erroneous counterexamples.
- Keeps the size of the abstract state space small to avoid state explosion problem.

CEGAR in Details

The initial abstract model is constructed using **existential abstraction** techniques

a **traditional** model checker determines whether ACTL* properties hold in the abstract model

YES
then the concrete model also satisfies the property.

NO
model checker generates a counterexample. It might not be valid (**spurious**)

CEGAR in Details (cont.)

- CEGAR provides a **symbolic algorithm** to determine whether an abstract counterexample is **spurious**.
  - If counterexample is not spurious:
    - it is reported to the user and model-checking stops.
  - If counterexample is spurious:
    - the abstraction function must be refined to eliminate it.

CEGAR guarantees to either **find** a valid counterexample

or

prove that the systems satisfies the desired property.

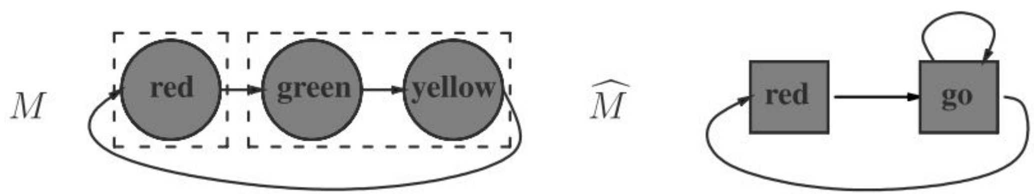
Example

- Assume that for a traffic light controller we want to **prove**:

$$\psi = \mathbf{AG}\,\mathbf{AF}(state = red)$$

- using the **abstraction function *h***:

$$h(red) = red \text{ and } h(green) = h(yellow) = go.$$



- We see $M \models \psi$ while $\widehat{M} \not\models \psi$.
  - infinite trace $\langle red, go, go, \ldots \rangle$ which <u>invalidates the specification</u> is a **spurious counterexample**

CEGAR Methodology

Given program P and and ACTL* formula φ our goal it to check whether the Kripke structure M corresponding to P satisfies φ.

1. **Generate the initial abstraction:** generating an initial abstraction h by examining the transition blocks corresponding to the variables of the program.
2. **Model-check the abstract structure:** checking $\widehat{M} \models \varphi$,
   a. if affirmative we conclude $M \models \varphi$.
   b. if reveals a counterexample $\widehat{T}$, we ascertain whether $\widehat{T}$ is an actual counterexamples
      i. if $\widehat{T}$ is an actual counterexample, then report it to the user, otherwise it is a spurious counterexample, and proceed to **STEP 3**.
3. **Refine the abstraction:** transforming the abstraction function to a more specific one
   a. after the refinement the abstract structure $\widehat{M}$ corresponding to the refined abstraction function does not admit the spurious counterexample $\widehat{T}$.
   b. after refining the abstraction function, return to **STEP 2**.

Advantages of CEGAR

1. The technique is complete for ACTL* specifications, i.e., it guarantees to either find a valid counterexample or prove that the system satisfies the desired property.

2. The initial abstraction and the refinement steps are efficient and entirely automatic. All algorithms are symbolic.

3. In comparison to other methods, CEGAR allows a finer refinement of abstract states.

4. The refinement procedure is guaranteed to eliminate spurious counterexamples while keeping the state space of the abstract model small.

Transition Blocks

- Each variable v_i in the program P has an associated **transition block** which defines the initial value and the transition relation for the variable.

$\mathbf{init}(v_i) := I_i;$	$\mathbf{init}(x) := 0;$	$\mathbf{init}(y) := 1;$
$\mathbf{next}(v_i) := \mathbf{case}$	$\mathbf{next}(x) := \mathbf{case}$	$\mathbf{next}(y) := \mathbf{case}$
$C_i^1 : A_i^1;$	$reset = \mathrm{TRUE} : 0;$	$reset = \mathrm{TRUE} : 0;$
$C_i^2 : A_i^2;$	$x < y : x + 1;$	$(x = y) \wedge \neg(y = 2) : y + 1;$
$\cdots : \cdots;$	$x = y : 0;$	$(x = y) : 0;$
$C_i^k : A_i^k;$	$\mathbf{else} : x;$	$\mathbf{else} : y;$
$\mathbf{esac};$	$\mathbf{esac};$	$\mathbf{esac};$

- $I_i \subseteq D_{v_i}$ is the initial expression for v_i
- C_i^j is a condition (a predicate)
- A_i^j is an expression
- Semantic of the transition block is similar to sematic of **case** statement in SMV

Identification of spurious path counterexample

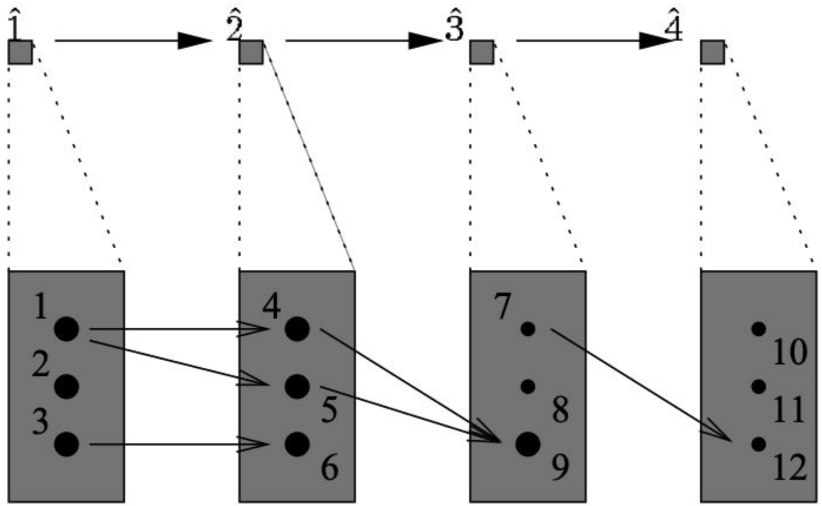


Fig. 3. An abstract counterexample

Algorithm SplitPATH

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

$j := 1$

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

$\quad j := j + 1$

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

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

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

else output j, S_{prev}

Fig. 4. SplitPATH **checks** spurious path.

Identification of spurious loop counterexample

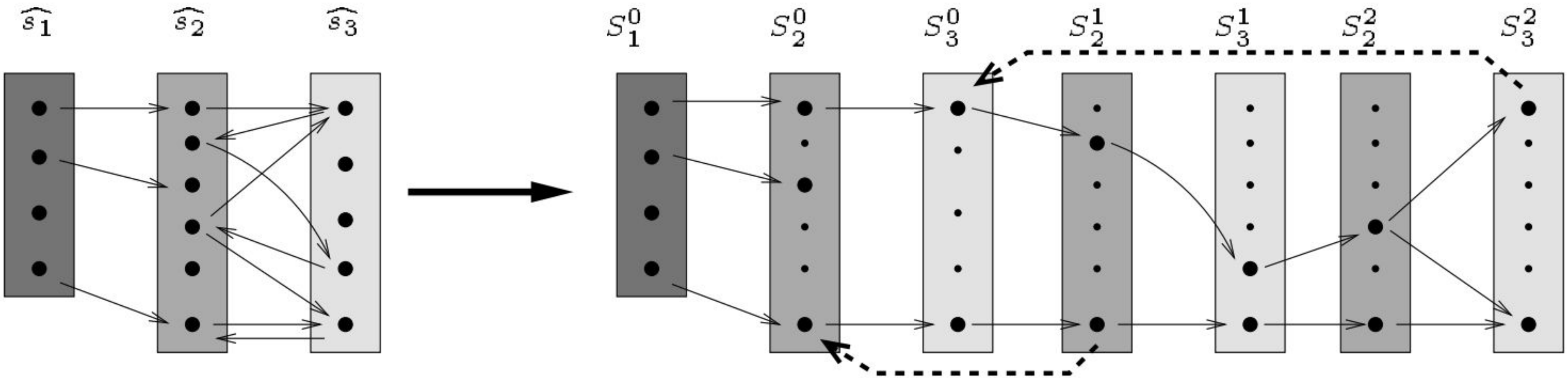


Fig. 5. A loop counterexample, **and** its unwinding.

Algorithm PolyRefine

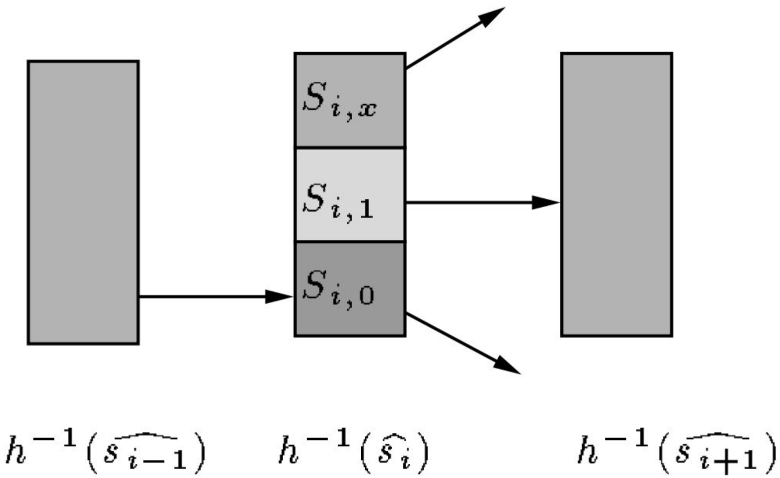


Fig. 6. Three sets $S_{i,0}$, $S_{i,1}$, and $S_{i,x}$

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**

Thank you