Working out an example with CEGAR

Shubham Sahai Srivastava
Indian Institute of Technology, Kanpur
ssahai@cse.iitk.ac.in

Abstract

In this article we will briefly introduce the abstraction refinement approach (CEGAR) [1] [2], developed by Clarke et. al.. CEGAR is an automatic iterative abstraction refinement technique for symbolic model checking. We will work with an example, and will demonstrate how the different steps of CEGAR algorithm work to generate an abstract model, which is free from the spurious counterexample encountered.

1. Introduction

In today's world software and hardware system are being used in all walks of life, ranging from our smart phones to the life critical pacemakers, from secure networking protocols to controllers of auto-mobile, aircraft and power grids. Due to this unprecedented mix of technology with our day to day lives, any defect in software and hardware system can prove to be fatal, as well as extremely costly. Hence, the need to effectively verify the correctness of the system is of utmost importance. There are different techniques which can be used for verification of the system, depending on which phase of the development life cycle the system currently is. These technique range from peer reviewing, to testing, simulation, and formal verification. In this article we will focus on the formal verification techniques to verify the systems, specifically the *model checking* technique.

Model checking is an automatic, model based, property verification approach which have been successfully used for past few decades to verify software and hardware systems. It verifies whether a system satisfies its specification by (i) representing the system as a finite Kripke structure, (ii) writing the specification in suitable temporal logic, and (iii) algorithmically checking whether the Kripke structure models the specification formula. But, in order to verify large systems, the major hurdle that one needs to overcome is the state explosion problem. State space abstraction is one of the several ways to handle the state explosion problem. It has proven successful in verifying large system designs, but, up till the year 2000, it was a manual process, requiring insight and creativity. There was an utmost need to automate this process of abstraction, in order to handle larger projects having industrial complexity.

In the year 2000 Clarke et. al. developed a technique to achieve the same (see [1], [2]). Counterexample guided abstraction refinement (CEGAR) is a technique, in which given a model ($\mathcal{M}$) of the system and a specification (φ), automatically generates an abstract model ($\widehat{\mathcal{M}}$), such that, $\mathcal{M} \vDash \varphi \Leftrightarrow \widehat{\mathcal{M}} \vDash \varphi$. The approach can be summarized as, given a specification and model of the system, generate an initial abstract model($\widehat{\mathcal{M}}$) for the corresponding model, and then iteratively refine it, using the spurious counterexample that gets generated while model checking ($\widehat{\mathcal{M}}$) for the specification φ. We essentially repeat this process until we find an abstraction, which satisfies our end goal (i.e. $\mathcal{M} \vDash \varphi \Leftrightarrow \widehat{\mathcal{M}} \vDash \varphi$).

In this article, we will briefly discuss this CEGAR technique (in Section 2) and then we will examine the working of CEGAR with the help of an example (in section 3). We will see how each step of the technique work using our example model and one spurious counterexample that exists in the model.

2. Overview of CEGAR

Model Checking is an automatic, model based, property verification approach. It is intended to be used for the formal verification of concurrent, reactive systems. In model checking we model the system as a finite state transition system, and the properties to be verifies in some suitable logic. We then apply the verification algorithm to verify whether the model satisfies the given property or not. It has been successfully applied for decades to verify hardware and software systems.

The process of model checking can be generalised as :

1. **Modelling System**

   Any system that need to be verified, can be fundamentally broken down into:

   - *Set of atomic propositions*
   - *Set of states*, which can be intuitively thought of as the screen shot of the system at any instant. It captures the state of the variables of the system at an instant.
   - *Transition relation*, which defines transition from one state to another.

   In model checking, the system that needs to be verified is formally represented as a finite **Kripke Structure.** It can be thought of as a directed graph, whose vertices represent the states of the system, with one or more states being initial states and the edges represent the transition between the states. Thus, formally a Kripke structure of a set A of atomic proposition can be defined as

   $$\mathcal{M} = (S, I, R, L)$$

   where,

   - S : set of states,
   - $I \subset S$: set of initial states,
   - $R \subseteq S^2$: set of transition relation
   - $L : S \to 2^A$: labelling function, which labels each state with the set of atomic proposition that are true at that particular state.

2. **Specification of Properties**

   We want capture the properties that the system should satisfy. These properties need to be specified in some appropriate logic. We have, propositional logic, predicate logic and other higher order logic at or disposal. But, we need to capture the notion of time, which can not be done by these simple logic. Hence, we switch our attention to temporal logic.

   Temporal Logic is the logic extended with the notion of time, such that, it can capture the future behaviour of the system. The truth value of a formula in Temporal Logic is defined with respect to a model. Also, a temporal formula is not statically true or false in a model. It can be true in some states and false in other states.

   CTL*, a sub class of branching time temporal logic, which is an extension of propositional logic obtained by adding path quantifiers (**A**, **E**) and temporal operators (**X**, **G**, **F**, **U**). ACTL* is a subset of CTL* where only the path quantifier **A** is used, and negation is restricted to the atomic formulas. An important feature of ACTL* is the existence of counterexamples.

3. **Model Checking Problem**

   Given a Kripke structure $\mathcal{M}$ and a specification ϕ in some temporal logic, the model checking problem is the problem of finding all states s, such that $\mathcal{M}, s \vDash \varphi$ and checking if the inittial states are among these.

A practical problem in model checking is the *state explosion problem*. Kripke structure represent the state space of the system under consideration. So, its size is exponential in terms of the description of the system. Hence, even for systems of small size, it is often not feasible to compute their Kripke Structure explicitly. There are several techniques to handle this state explosion problem, like the use of BDD and symbolic model checking, bounded model checking, partial order reduction. But, in this article we will focus on Abstraction techniques and CEGAR (counter example guided abstraction refinement).

2.1 Abstraction

The main goal of abstraction is to reduce the state space by some means. The basic idea is to remove the irrelevant details and simplify the components. This would generate an abstract model corresponding to a concrete model. Now, as the abstract model is smaller, hence it would be easier to verify. But, naturally abstraction comes with information loss. An *over approximation* of concrete space, leads to false negatives. On the other hand, an *under approximation* leads to false positives.

Intuitively speaking, abstraction partitions the concrete state space of the Kripke structure into clusters, and treat these clusters as new abstract space. Formally, an abstraction function h is defined as a surjective mapping $h : S \to \hat{S}$, where S is the set of concrete states and $\hat{S}$ is the set of abstract space. Also, this surjection h induces an equivalence relation $\equiv_h$ on the domain S, in the following manner. Let $d, e \in S$, then

$$d \equiv_h e, \text{ iff } h(d) = h(e)$$

The corresponding abstract Kripke structure generated by abstraction function h, is given by

$$\hat{\mathcal{M}} = (\hat{S}, \hat{I}, \hat{R}, \hat{L})$$

where,

- $\hat{S}$: set of abstract states,
- $\hat{I}(\hat{d})$ iff $\exists d \in I, h(d) = \hat{d}$,
- $\hat{R}(\hat{d_1}, \hat{d_2})$ iff $\exists d_1, d_2 \in S$ such that, $h(d_1) = \hat{d_1} \wedge h(d_2) = \hat{d_2} \wedge R(d_1, d_2)$,
- $\hat{L}(\hat{d}) = \cup_{h(d)=\hat{d}} L(d)$.

An abstraction function h is said to be *appropriate* for a specification φ if none of the sub formulas of φ can differentiate two equivalent states $d, e \in S$, *i.e.*, for all sub formulas f of φ and $\forall d, e \in S$, we have :

$$d \equiv_h e \Rightarrow (d \vDash f \Leftrightarrow e \vDash f)$$

.

For an abstract state $\hat{d}$, we say $\hat{L}(\hat{d})$ is *consistent*, if all the concrete states corresponding to $\hat{d}$, have same label. Consequently, $\hat{L}(\hat{d}) = L(d)$, where d is some concrete state corresponding to $\hat{d}$.

The following theorem captures the one sided correctness of the abstract model.

Theorem 2.1. *Let h be appropriate for ACTL* specification φ. Then $\mathcal{M}' \vDash \varphi \Rightarrow \mathcal{M} \vDash \varphi$.*

But, we cannot directly claim that $\mathcal{M}' \nvDash \varphi \Rightarrow \mathcal{M} \nvDash \varphi$. This is because, existential approximation leads to false positives. As a result of which, the counterexample generated might be spurious, that was introduced by the abstraction, but, does not exist in the original model. Hence, we need to detect if the counterexample is spurious, and refine the model if it is spurious. In the next section we discuss an automatic technique to do the same.

2.2 Counterexample Guided Abstraction Refinement : CEGAR

CEGAR is an efficient, automatic abstraction refinement technique, where we begin with small abstract representation of the system to be verified, which essentially preserves the control structure of the program. This representation is referred as the *initial abstraction* of the system. We model check this abstract model, if no bug is found, we can conclude that the concrete model is also safe and exit.

But, if any counterexample is found, it might have been generated due to the over-approximation of the system. We check whether this counterexample is valid or not. To do this, we try to find the trace of the concrete states, corresponding to the abstract states in the counterexample. If this counterexample corresponds to the concrete trace, we conclude that the counterexample is *concrete* and the model does not satisfy the specification. On the other hand, no concrete trace exists corresponding to the counterexample, we call such counterexamples as *spurious*.

If the spurious counterexample is found, we use it to *refine* the abstract model in hand, to generate a finer abstraction, in which we get rid of this counterexample. We repeat this process, until we can conclude that the system satisfies the specification, by getting rid of all the counterexamples. Or, we conclude that the system does not satisfy the specification, by generating a concrete counterexample. Next, we briefly describe the individual steps of CEGAR.

1. **Generating initial abstraction :**

   Given a program $\mathcal{P}$, its corresponding model $\mathcal{M}$ and specification φ, we want an initial abstraction function h, such that,
   (a) h is appropriate for φ, and
   (b) h preserves the control flow skeleton of $\mathcal{P}$.

   For a set of formulas $\mathcal{F}$, we say two states are $\mathcal{F}$*-equivalent*, if we cannot distinguish them by the formulas in $\mathcal{F}$ *i.e.* $\forall d, e \in S, \forall f \in \mathcal{F}$, $d \vDash f \Leftrightarrow e \vDash f$. It can be shown that if $\mathcal{F} \supseteq Atoms(\varphi)$, then the corresponding abstraction function $h_{\mathcal{F}}$ is appropriate for φ, where, the function $Atoms()$ return the set of atomic formulas.

   Combining the observations above, we can define the *initial abstraction function* (init) as $h_{Atoms(\mathcal{P})}$. It can be shown that *init* is appropriate for φ.

   Next, we examine, *how to generate* the initial abstract model, corresponding to a given program $\mathcal{P}$. Let V be the set of variables in $\mathcal{P}$, and, $F = Atoms(\mathcal{P})$. Then,

   - We begin by partitioning F into the set of *Formula Clusters*, such that no two formula clusters share common variables.
   - The set of Formula clusters, induce a natural partition on the set of variables V, known as *variable clusters*.
   - For each of these variable clusters, we define an abstraction function, such that the abstract states are *consistent* with the cluster formulas.
   - The Cartesian product of all cluster abstractions, will give us our final abstraction.

2. **Checking for spurious counterexample :**

   We can generate the initial abstract model, corresponding to the initial abstraction function from the previous section. We model check this abstract model $\widehat{\mathcal{M}}$, to determine if $\widehat{\mathcal{M}}$ satisfies the specification φ. If it does, then we can conclude that the original model $\mathcal{M}$ will also satisfy φ. On the other hand, if it doesn't, then the model checker produces a counterexample. In this section we will examine the counterexample, and check if it is *concrete* or *spurious*.

For the sake of simplicity, we will first handle the case, when the counterexample $\widehat{T}$ is given by a path $\langle \widehat{s_1}, \cdots, \widehat{s_n} \rangle$. The intuitive idea is to generate the set of paths in the concrete model, corresponding to the given path $\widehat{T}$ in the abstract model. We can define a function $h^{-1}(\widehat{s}) := \{s \mid h(s) = \widehat{s}\}$, and, we can further extend h^{-1}, to the set of paths as :

$$h^{-1}(\widehat{T}) == \left\{ \langle s_1, \cdots s_n \rangle \;\middle|\; \bigwedge_{i=1}^{n} h(s_i) = \widehat{s_i} \wedge I(s_1) \bigwedge_{i=1}^{n-1} R(s_i, s_{i+1}) \right\}$$

Clearly, for a concrete counterexample, the above set $h^{-1}(\widehat{T})$ is not empty. Further, we can define the set of reachable states corresponding to the abstract state in $\widehat{T}$, $S_1 = h^{-1}(\widehat{s_i}) \cap I$, and $\forall i, S_i := Img(S_{i-1}, R) \cap h^{-1}(\widehat{s_i}$, where $Img(S_{i-1}, R)$ is the forward image of S_{i-1} with respect to transition relation R. It can be shown that, if $\widehat{T}$ is a concrete counterexample, then $\forall i \in [1, n],\ S_i \neq \phi$.

Algorithm 1 utilizes this observation to check if the counterexample is concrete or spurious. In case the counterexample is concrete, the algorithm outputs "Counterexample exists", otherwise, it returns the index j for which $S_j = \phi$, and the *failure state*, which is the previous state S_{prev} corresponding to S_j. We represent S_{prev} as failure state, as there were no outgoing transitions from S_{prev} to S_j.

Algorithm 1 : splitPATH($\widehat{T}$, $\mathcal{M}$, $\widehat{\mathcal{M}}$)

Require: Counterexample $\widehat{T}$ having length n, and models $\mathcal{M}$, $\widehat{\mathcal{M}}$

```
S ← h^{-1}(ŝ_1) ∩ I
j ← 1
while S ≠ φ and j < n do
    j ← j + 1
    S_prev ← S
    S ← Img(S, R) ∩ h^{-1}(ŝ_j)
if S ≠ φ then
    return "Counterexample exists"
else
    return j, S_prev
```

If the counterexample is found to be *spurious*, we move to the *refinement* step, to refine our abstract model in such a way, that we get rid of this counterexample in the new abstract model.

Remark. *In case the counterexample is a loop, we can reduce to problem to path counterexamples, by unwinding the loop ω number of times, where, ω is the size of the smallest abstract state occurring in the loop.*

3. **Abstraction refinement :** In the previous section, if the counterexample turns out to be a spurious one, then we need to refine our model, such that in the new refined model, this counterexample is eliminated.

Examining the failure state in some more detail, we can classify the set of concrete states corresponding to the abstract state as for a counterexample :

- *Dead-end* state (S_D) : The set of states which are reachable from the initial states, but do not have any outgoing edges to the next state.
- *Bad* state (S_B) : The set of states that have outgoing edges to the next state, but are not reachable from the initial state.
- *Irrelevant* state (S_I): All the other states.

Its the coexistence of the dead-end states and the bad states in the same state, that results in generating spurious counterexample. Hence, we want to separate S_D and S_B in our refined model. It can be seen clearly, that doing this will eliminate this counterexample in this refined model.

We define the projection set, for a given set $X \subseteq S$, an index j, and an element $a \in D_j$ as : $Proj(X, j, a) = \{(s_1, \cdots, s_{j-1}, s_{j+1}, \cdots, s_m) | (s_1, \cdots, s_{j-1}, a, s_{j+1}, \cdots, s_m) \in X\}$. It can be shown as a *necessary* condition, that for two states $d, e \in S$, if $Proj(S_D, j, d) \neq Proj(S_D, j, e)$ then every refinement must separate d and e. Algorithm 2 utilizes this property of projection function, to refine the abstract model.

Algorithm 2 : polyRefine

```
1: for j ← 1 to m do
2:     ≡'_j := ≡_j
3:     for every a, b ∈ E_j do
4:         if proj(S_D, j, a) ≠ proj(S_D, j, b) then
5:             ≡'_j := ≡'_j \ {(a, b)}
```

3. Working Example

Let us consider a *hypothetical scenario* of a chemical factory, where two chemicals C_1 and C_2 are being mixed in an insulated tank. The chemicals being highly reactive, we want to regulate their respective temperature, before they are added to the tank for mixing. The temperature difference between chemicals should be at least 60°C at all times. Also, the chemicals themselves should be in their respective recommended temperature ranges : $20°C - 22°C$ for chemical C_1 and $80°C - 82°C$ for chemical C_2. Whenever the temperature of the chemicals, reach there respective maximum limit, we need to stop the reaction, reset the temperature to their lower limit and start the reaction again. If at any time, both the chemicals are $2°C$ above their prescribed range, the reaction could be highly explosive and we would need to evacuate the facility and shut it down to ensure safety.

We would want to have a controller, which automatically regulates the temperatures of the two chemicals, and ensures that they are always in their respective safety limits. Basically, we do not want to encounter a situation, where we have to evacuate the facility. The code presented in Figure 1, claims to handle this situation. We want to formally verify the claim.

3.1 Modelling the system

The given example (Figure 1) presents the C code for the controller. As the first step, we would model the system, to obtain the corresponding Kripke structure and then formally present the specification in suitable temporal logic.

```
1  void main() {
2      int t1 = 20;                    // Temperature corresponding to chemical 1
3      int t2 = 80;                    // Temperature corresponding to chemical 2
4      while(1) {
5          if(t2-t1 == 60) {
6              if (t2 != 82) {
7                  t1 = 20;
8                  t2 ++;
9              }
10             else if (t2 == 82) {
11                 t1 = 20;
12                 t2 = 80;
13             }
14         }
15         if (t2 - t1 > 60) {
16             t1++;
17         }
18         if (t1 == 24 && t2 == 84) {
19             EvacuateFacility();     // Auxiliary code for invoking security measures
20         }
21     }
22 }
```

Figure 1: *Program ($\mathcal{P}$) for handling the hypothetical scenario of the chemical factory described above*

1. **Generating Kripke Structure :**

   Program $\mathcal{P}$ given above, comprises of 2 variables $t1$ and $t2$, having Integer domain. But, in the program above, not all Integer values can be achieved. So we can restrict the domain of the variable to the values of interest. Hence, the domain of variable $t1$ is given by $D_{t1} = \{20, 21, 22, 23, 24\}$ and domain for $t2$ is given by $D_{t2} = \{80, 81, 82, 83, 84\}$. So, the state space for $\mathcal{P}$ would be $S = D_{t1} \times D_{t2}$. The set of atomic propositions in $\mathcal{P}$ is given by $A = \{(t2 - t1 = 60), (t2 = 82), (t2 - t1 > 60), (t1 = 24), (t2 = 84)\}$. The Kripke structure $\mathcal{M}$ corresponding to $\mathcal{P}$ can be defined as :

   $$\mathcal{M} = (S, I, R, L)$$

   where,

   - $S = D_{t1} \times D_{t2}$,
   - $I = \{(20, 80)\}$,
   - $R \subseteq S^2$: set of transition relation (show in Figure 2),
   - $L : S \to 2^A$: labelling function, which labels each state with the set of atomic proposition from A, that are true at that particular state.

   Remark. *The set of transition relation R can be obtained from the control structure of the given program $\mathcal{P}$ specified is lines $5 - 17$, i.e. by considering any state $(t1, t2)$, we can directly obtain the next state $(\widehat{t1}, \widehat{t2})$ from line lines $5 - 17$ in $\mathcal{P}$.*

2. **Specification :** We want to formally specify the property, that we want to be true in $\mathcal{P}$. As, discussed above, we do not want to reach a situation, where we would have to evacuate the facility. From the code above, we can see clearly, that the function *EvacuateFacility()*, is called only when $t1 = 24$ and $t2 = 84$. Hence, we can state the property that needs to be true as : It should never be the case that $t1 = 24$ and $t2 = 84$.

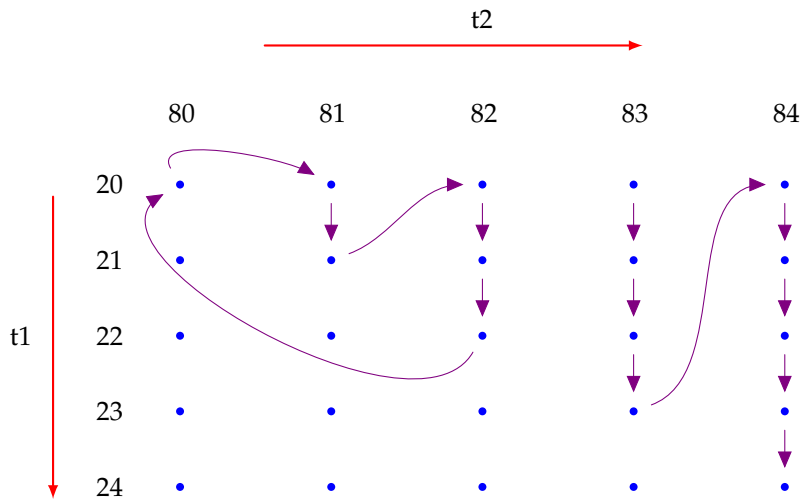

Figure 2: *Transition relation R for program $\mathcal{P}$*

Formally, we can represent this property in ACTL* as : $\varphi := \mathbf{AG}\Big(\neg\Big((t1 = 24) \wedge (t2 = 84)\Big)\Big)$.

3.2 Applying CEGAR

In the previous section, we had constructed the model of the given program $\mathcal{P}$. In this section, we will apply the CEGAR technique on the given model $\mathcal{M}$, in order to generate a corresponding abstract model $\widehat{\mathcal{M}}$, such that $\mathcal{M} \vDash \varphi \Leftrightarrow \widehat{\mathcal{M}} \vDash \varphi$). We will follow the outline presented in Section 2.2.

1. **Generating Initial Abstraction :**

   We have the model $\mathcal{M}$ of the program $\mathcal{P}$ and our aim is to automatically generate the initial abstraction of $\mathcal{M}$. The set of atomic propositions in $\mathcal{P}$ is $A = \{(t2 - t1 = 60), (t2 = 82), (t2 - t1 > 60), (t1 = 24), (t2 = 84)\}$. We can clearly see, that the entire set A would form a single *Formula Cluster*, as every formula *interferes* (have atleast one common variable) with atleast one of the formula. This results in a single *variable cluster*, i.e. $\{t1, t1\}$.

   Next, we want to generate the *equivalence classes* for the set S, with respect to the formula cluster A. Note, that any two pair of states correspond to the same equivalence class, iff they cannot be distinguished by the atomic formulas of the formula cluster A. We give the equivalence classes in Table 1 :

   The initial abstraction function *init*, is given by h_A, which maps any two concrete states $d, e \in S$, to the same abstract states, if we can not distinguish d and e by the formulas in A, *i.e.* $d \vDash f$, *iff* $e \vDash f, \forall f \in A$. Table 1 gives the mapping generated by this initial abstraction function, h_A, and the corresponding equivalence classes.

   Now, for the abstract model $\widehat{\mathcal{M}}$, can be formally given by: $\widehat{\mathcal{M}} = \{\widehat{S}, \widehat{I}, \widehat{R}, \widehat{L}\}$, where,

   - $\widehat{S} = \{\widehat{0}, \widehat{1}, \widehat{2}, \widehat{3}, \widehat{4}, \widehat{5}, \widehat{6}, \widehat{7}, \widehat{8}, \widehat{9}\}$,
   - $\widehat{I} = \{\widehat{0}\}$.

Abstract state	Concrete states	Labels for concrete states
$\widehat{0}$	$\{(20,80),(21,81),(23,83)\}$	$\{(t2-t1=60)\}$
$\widehat{1}$	$\{(20,81),(20,83),(21,83),(22,83)\}$	$\{(t2-t1>60)\}$
$\widehat{2}$	$\{(20,82),(21,82)\}$	$\{(t2=82),(t2-t1>60)\}$
$\widehat{3}$	$\{(22,82)\}$	$\{(t2-t1=60),(t2=82)\}$
$\widehat{4}$	$\{(20,84),(21,84),(22,84),(23,84\}$	$\{(t2-t1>60),(t2=84)\}$
$\widehat{5}$	$\{(24,84)\}$	$\{(t2-t1=60),(t1=24),(t2=84)\}$
$\widehat{6}$	$\{(23,82)\}$	$\{(t2=82)\}$
$\widehat{7}$	$\{(24,82)\}$	$\{(t1=24),(t2=82)\}$
$\widehat{8}$	$\{(24,80),(24,81),(24,83)\}$	$\{(t1=24)\}$
$\widehat{9}$	$\{(21,80),(22,80),(23,80),(22,81),(23,81)\}$	$\{\}$

Table 1: *Initial abstraction function*

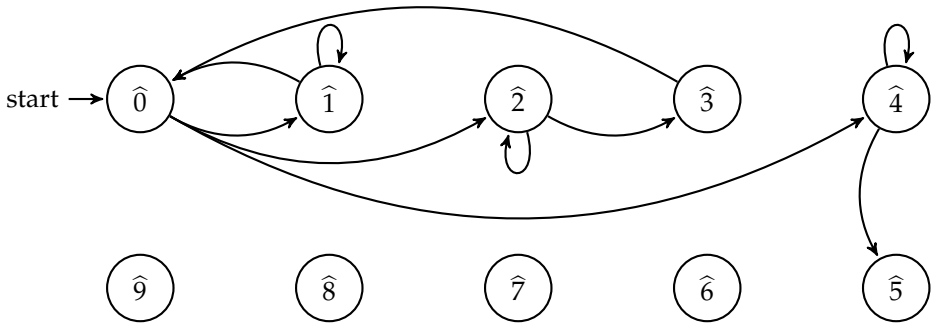

Figure 3: *Transition relation $\widehat{R}$ corresponding to abstract model $\widehat{\mathcal{M}}$*

- The transition relation $\widehat{R}$ is shown in Figure 3.
- The labels for each state in $\widehat{\mathcal{M}}$, is given by the 3^{rd} column in Table 1.

2. **Checking for spurious counterexample :**

   We have the abstract model $\widehat{\mathcal{M}}$ and the specification φ for the program $\mathcal{P}$. The next step is to model checking for this abstract model. Model checking would either claim that the $\widehat{\mathcal{M}}$ models φ, in which case we conclude that $\mathcal{M}$ also models the specification φ. On the other hand, it might generate a *counterexample*, which if found to be *spurious*, would mean we need to refine our model, and if found to be a correct one, would mean that the original system also does not models the specification. In this section, we will examine one of the generated counterexample, and will decide if that counterexample is spurious or a concrete one.

   In our example, the specification would be False, if we have to evacuate the facility. It can be seen clearly, that the function *EvacuateFacility()*, would be called iff, $t1 = 24$ and $t2 = 84$. Hence, we can say φ is False iff state $\widehat{5}$ is reachable in the abstract model $\widehat{\mathcal{M}}$. Clearly, from the transition diagram given in Figure 3, we can see that $\widehat{5}$ is reachable. Hence, as of now,

for the model $\widehat{\mathcal{M}}$, the specification φ is not True. Let us consider, one of the counterexample that could be generated, $\widehat{T} = \{\widehat{0}, \widehat{1}, \widehat{0}, \widehat{2}, \widehat{3}, \widehat{0}, \widehat{4}, \widehat{5}\}$. In this section we will apply Algorithm 1 $\Big($**splitPATH**$(\ \widehat{T}$**,** $\mathcal{M}$ **,** $\widehat{\mathcal{M}}\ )\Big)$, on this counterexample, and will examine if $\widehat{T}$ is a concrete or a spurious counterexample.

Step 1 of the algorithm, gives us the concrete initial states from the model $\mathcal{M}$, corresponding to the first state of the counterexample (in this case, the abstract state $\widehat{0}$). For $\widehat{T}$, from step 1, we get $S = \{(20, 80)\}$.

Next, we enter the loop in steps $3 - 6$. The subsequent set of concrete states, reachable from the initial states are given by : $S_1 = \{(20, 81)\}$, $S_2 = \{(21, 81)\}$, $S_3 = \{(20, 82), (21, 82)\}$, $S_4\{(22, 82)\}$, $S_5 = \{((20, 80)\}$, $S_6 = \phi$. The loop exits, as we have encountered a state $S_6 = \phi$, and the algorithm returns $j = 7$, and the *failure state* : $\widehat{0}$ (which was the present value of S_{prev}).

From this output of the algorithm **splitPATH**$(\ \widehat{T}$**,** $\mathcal{M}$ **,** $\widehat{\mathcal{M}}\ )$, we can conclude the following :

(a) The counter example $\widehat{T}$ is *spurious* and we need to go to the *refinement* step, to refine our abstract model $\widehat{\mathcal{M}}$

(b) The *failure state* in the model is the state $\widehat{0}$, *i.e.* this state comprises of both *dead-end* states and *bad* states.

(c) The edge between the 6^{th} and the 7^{th} node in the counterexample $\widehat{T}$, does not correspond to any edge in the concrete model.

3. **Abstraction Refinement :**

In the previous section we have found that the counterexample $\widehat{T}$ is the spurious one. In this section, using $\widehat{T}$ and the *failure state* $(\widehat{0})$ returned by the algorithm **splitPATH**$(\ \widehat{T}$**,** $\mathcal{M}$ **,** $\widehat{\mathcal{M}}\ )$, we will refine our abstract model, to generate a new model in which this spurious counterexample is eliminated. After that we will again model check the new abstract model and repeat the above procedure.

In our case, the failure state is $\widehat{0} = \{(20, 80), (21, 81), (23, 83)\}$. We want to identify the set of *dead-end* states and *bad* states corresponding to this failure state. Using figure 2, we can see that, here the corresponding set $S_D = \{(20, 80), (21, 81)\}$, as they are reachable from the initial state and do not have any outgoing edge to the next state *i.e.* $\widehat{4}$. Also, $S_B = \{(23, 83)\}$, as this state is not reachable from the initial state, but has an outgoing edges to the next state *i.e.* $\widehat{4}$.

Applying algorithm 2 (polyRefine), we separate S_D and S_B from $\widehat{0}$, thereby generating two different states, $\widehat{0}'$ and $\widehat{0}''$. The new abstract model generated after splitting $\widehat{0}$, is shown in the Kripke Structure in Figure 4.

As it can be seen clearly in the Figure 4, that there still exists a path from initial state to the state $\widehat{5}$. Hence, the new refined model still does not satisfy the specification φ. Hence, model checking this refined model, will still generate a counter example and we will need to verify whether this new counterexample is concrete or spurious. Hence, we need to repeat Step 2 and 3 again, until, we are free from counterexample and could claim that the original model satisfies the specification φ, or we generate a counterexample and hence, claim the the original model does not satisfy the specification φ.

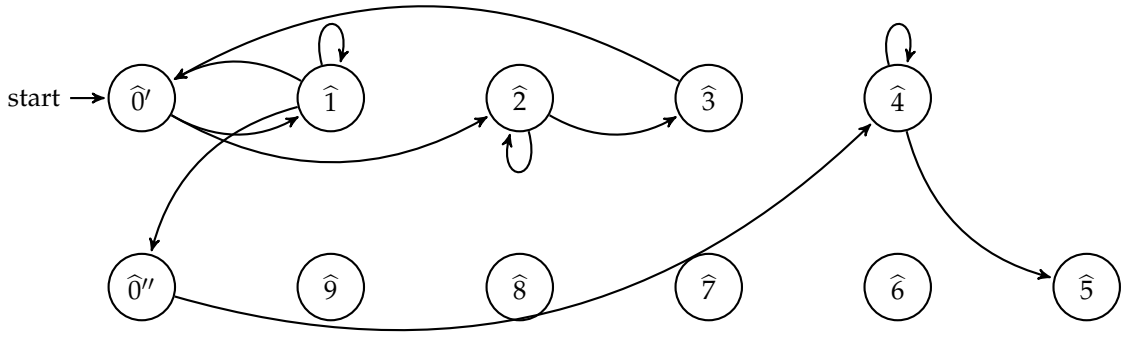

Figure 4: *New transition relation $\widehat{R}'$ corresponding to the new abstract model $\widehat{\mathcal{M}}$, obtained after splitting the state $\widehat{0}$ into two different states $\widehat{0}'$ and $\widehat{0}''$.*

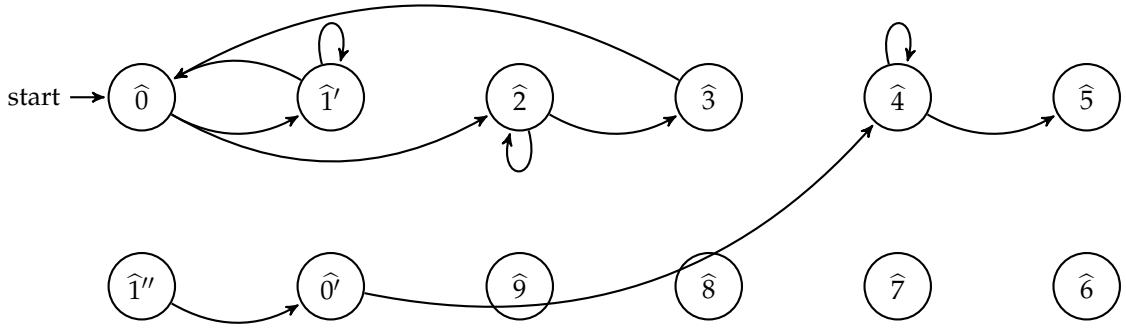

Figure 5: *Transition relation $\widehat{R}$ corresponding to final abstract model $\widehat{\mathcal{M}}$*

4. Conclusion

In this article, we gave overview of Counterexample guided abstraction refinement technique, introduced by Clarke et.al. [1]. We also worked out an example, where we examined how various steps of CEGAR work in order to generate a refined abstract model.

As discussed in previous section, the abstract model obtained in Figure 4 still does not satisfy the specification. We can clearly see a path from initial state to the state $\widehat{5}$. But, as it turns out, even this path is spurious. On applying the above steps of CEGAR again on any counterexample, we will get the state $\widehat{1}$ as the failure state.

The final refined model is shown in Figure 5. As we can see clearly, in this final model, there is no path from initial state to $\widehat{5}$. Hence, this final model, satisfies the specification φ. Consequently, the original model $\mathcal{M}$ also satisfies the specification φ.

References

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," *Computer Aided Verification: 12th International Conference, CAV 2000, Chicago, IL, USA. Proceedings*, pp. 154–169, July 2000.

[2] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement for symbolic model checking," *J. ACM*, vol. 50, pp. 752–794, September 2003.