Static Checking By Means of Abstract Interpretation

Kaninda Musumbu
LaBRI (UMR 5800 du CNRS),
Université Bordeaux 1, France
351, cours de la Libération, F-33.405 TALENCE Cedex
e-mail: musumbu@labri.fr

Abstract

Dynamic checking are usually easier to use, because the concept are established and wide well know. But they are usually limited to systems whose state space is finite. In an other part, certain faults cannot be detected dynamically, even by keeping track of the history of the state space.Indeed, the classical problem of finding the right test cases is far from trivial and limit the abilities of dynamic checkers further. Static checking have the advantage that they work on a more abstract level than dynamic checker and can verify system properties for all inputs . Problem, it is hard to assure that a violation of a modeled property corresponds to a fault in the concrete system. We propose an approach, in which we generate counter-examples dynamically using the abstract interpretation techniques.

1 Introduction

Being given that the number of state of a model believes in an exponential way with the number of variables and components of the system, the model-checking became complicated to treat in an automatic way. In order to make this work realizable, it is necessary to reduce the sizes of these models with an aim of reaching time and reasonable memory capacities. The techniques of reduction seek to suppress the harmful effects of the combative explosion. When the graphs of behavior comprise several million or milliards of states and transitions, the physical limits of the memory are quickly reached. It is then necessary to resort to techniques compressions of the graphs of behavior. Most known is based on the BDD (Binary Decision Diagrams). At the enumeration time, to decide if a reached state was already met requires to traverse the explored part of the graph. This subgraph, which does not cease growing bigger, must be arranged in the read-write memory. The limits of this memory are quickly exceeded and the implementation of algorithms of pagination know a considerable fall of performances. The methods of abstraction make it possible to eliminate the proliferation from different states(ones from the other) by possibly unimportant details within sight of the properties to be checked. It is essential that the small-scale model preserve sufficient information to produce the same results as the models of origin and to preserve the same properties that one wishes to check. These two exigences must be considered with attention at the time of the generation of a abstract model starting from a concrete model. To conceive a "good" method of reduction consists to produce a reduction relation verifying three criteria: an important reduction ratio, a relation of strong preservation and an easy deduction of the relation of reduction starting from the description of the system, the ideal being the construction of the reduced graph directly starting from the description. The way whose details of the abstraction will be selected for the checking can be made in an automatic or manual way. The manual technique includes abstract interpretations selected by the user. The abstractions considered generally preserve the properties in a weak way, which means that they are only preserved abstracted model with the concrete model. Thus, if one can guarantee that a property is checked, that is different with its negation. The abstract interpretation is a methodology aiming at defining, analyzes and justifier your techniques of approximate computation of properties of systems [**?**]cc).Whatever the semantics may be used. It then consists in placing the analysis not in the concrete domain but in a abstract domain, (simplified and limited) which conserves the search properties, the major disadvantage is that the results are in general less precise and that one needs accommodate approximations of the properties. In our paper, we present a technique of abstraction called *abstraction by predicate* of a refinement to reduce the generality and the minimality of the analysis, thus a violation of a property detected on one of abstract path has a strong probability of existing on a path of the concrete model.

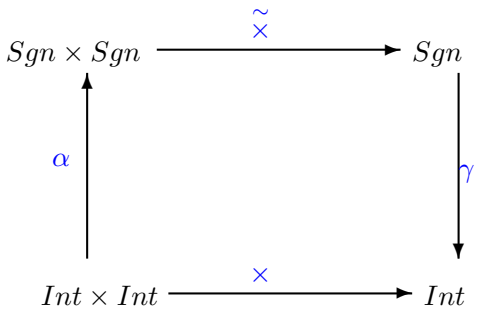
978-0-7695-3308-7/08 $25.00 © 2008 IEEE
DOI 10.1109/ICCSIT.2008.10

2 Generality

Small Example: The rule of Signs

$\tilde{\times}$	$+$	0	$-$
$+$	$+$	0	$-$
0	0	0	0
$-$	$-$	0	$+$

Such that the following diagram commutates



Consistency (soundness):

$$\forall x, y \in Int : X \times y \in \gamma(\alpha(x) \tilde{\times} \alpha(y))$$

2.1 Definition

Abstract Interpretation is a general methodologies for automatic analysis of the run-time properties of system.. The problem is that the exact analysis may be very expensive, sometimes through decidable properties may be NP-complete. The idea is to find a decidable approximation which is soundness and calculable.

2.2 Mathematical theory of Abstract Interpretation

Often, AI refers to the concept of *connection galoisienne* a 4-tuple (C, A, α, γ) where C and A are complete lattices , $\gamma : A \to C$ and $\alpha : C \to A$ are monotonous functions such as:

$$\begin{array}{lll} \forall \beta \in a : & \alpha(\gamma(\beta)) & = \beta, \\ \forall C \in C : & \gamma(\alpha(c)) & \geq C. \end{array}$$

Impossible into practice of *generating and analyzing* all possible traces of execution for a given program .

2.3 Motivation

Abstract interpretation is based on three fundamental ideas :
abstract domain, abstract operators and point fixes computation Abstract domain and abstract operators are used to carry out a program on abstract values. Computation of the fixed point: directs the process on abstract values (define in a certain way the semantics of the program) Objective: To obtain information on the execution and the results of program. Provided that the abstract domain and operators satisfy certain constraints.

2.4 Semantics

Its definition has two view points:

- **Theoretical** associates a meaning to objects handled by the programs.
- **Piratical** associates a program a semantic function (stomata).

$$< \mathcal{P}, e > \xrightarrow{\tau} < \mathcal{P}', e' >$$

$$\mathcal{P} = \tau.\mathcal{P}' \text{ and } e' = \tau(e)$$

τ is a transition related. Note:

$$\tau[\mathcal{P}](e_0) = cal$$

if $cal =< \_, e_n >$ then the results of the program are the values of variables in the last state.

- **Denotational**

$$\tau : (D \longrightarrow D) \longrightarrow (D \longrightarrow D)$$

$$\tau = \lambda f.\lambda x.(\text{ if } p(x) \text{ then } x \text{ else } f(h(x)) \text{ fi}$$

where p is a predicate , h any function. Example F91McCarthy:

$$\tau = \lambda f.\lambda x.(\text{ if } x > 100 \text{ then } x{-}10 \text{ else } f(f(x{+}11) \text{ fi}$$

2.4.1 Abstract Domain

: Any program P handles data which belong to a D_s domain says standard. To make abstract interpretation will consist in choosing an abstraction of data D_{abs} First Approach

$$\theta = \{x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n\}$$

β approximates θ iff

$$\beta = \{x_1 \leftarrow prop(t_1), \ldots, x_n \leftarrow prop(t_n)\}$$

this define concretes semantics
More often no-calculable.
Construction process P can be consider like a partial function of

$$P : D_s^m \longrightarrow D_s^n, n, m0$$

Example: function of McCarty known as function91

$F91(x) =$ if $x > 100$ then $x - 10$ else $F91(F91(x + 11))$

```
int F91(int x)
     int F;
          if (x >100) f =x-10;
          else f=F95(F95(x+11));
     return F;
```

```
int F91McCarthy(void){
   int x;
   scanf(&x);
   printf("value of F91 of %d = %d ",
           x, F95(x));
   exit(0);
}
```

Note:: There is not a proof of termination of $F91McCarthy, \forall X \in \mathcal{Z}$
The idea is to replace Z by its power set $\mathcal{P}(\mathcal{Z})$.
We get the following definition:

$$F91(X) = \{x - 10 : x > 100 \wedge x \in X \subseteq Z\} \cup F91(F91(\{x + 11 : x \leq 100 \wedge x \in X \subseteq Z\}))$$

It is easy to show that $F91(C_1) = C_2$ verifies the condition: $\forall x \in C_1 \exists y \in C_2 : y = f(x)$

Note:: the calculation of such function is too expensive for simple value, the definition of the operations on a such domain is too complex.

Second Approach
To choose "a good" system of representation of properties

$$\beta = \{x_1 \leftarrow \alpha(prop(t_1)), \ldots, x_n \leftarrow \alpha(prop(t_n))\}$$

Choice of an (judicious) approximation of each element of $\mathcal{P}(\mathcal{Z})$ by an interval $[min..max]$

$$D_{abs} = \{[s..t] : s, t \in Z \cup \{-\infty, +\infty\}\}$$

we define an order on D_{abs}, noted $\subseteq$: $[s..t] \subseteq [s'.t']$ iff $s \geq s'$ $t \leq t'$

Lemma: $(D_{abs}, \subseteq)$, is a lattice whose lower bound is [] and the upper bound is $[-\infty, +\infty]$. Abstraction and concretization function:

$$\alpha : C \longrightarrow A : C \rightarrow \alpha(c) = [min(c)..max(c)]$$

$$\gamma : A \longrightarrow C : a \rightarrow \gamma(a) = [s, s + 1..., t - 1, t]$$

with $a = [s..t]$ such that they verifying the constraints of coherence:

$$\forall c \in C : \gamma(\alpha(c)) \supseteq c \forall a \in A : \alpha(\gamma(a)) = a.$$

Remark

- an equivalent abstract of a program carries out the same standard operations that the original except that the domains are different.
- for a real Pascal, C or Java programs, the work of rewrite would be too tiresome. In fact one defines abstracted operators, the abstract interpretor uses those to carry out calculations on the abstract data by interpreting the program to be analyzed.
- In practice each operator or function of the language must have an abstract equivalent. The quality required is their consistency, their coherency with respect to their equivalent concrete operator. For the reason of performance, one requires the efficiency and convergence to guarantee a termination and acceptable computing time.

Abstract version of the F91 function:

$$\begin{aligned} F_a91([s..t]) = \quad & [max(91, s - 10)..(t - 10)] \cup \\ & F_a91(F_a91([(s + 11)..min(t + 11, 111)])) \\ \forall I_i, I_j \in D_a : \quad & I_i = [s..t], I_j = [s'..t'] \Rightarrow \\ & I_i \cup I_j = lub(I_i, I_j) = [min(s)..max(t, you)], \end{aligned}$$

The abstract calculus:

$$F_a91([-\infty..+\infty]) = [91, +\infty] \cup F_a91(F_a91([-\infty.., 111]))$$

$$F_a91([-\infty..111]) = [91, 101] \cup F_a91(F_a91([-\infty.., 111]))$$

Note:: The set of functions of $D_a \longrightarrow D_a$ can be provided with an order $f \leq G$ iff $\forall I \in D_a : f(I) \subseteq g(I)$
The fixpoints calculus: it is useful at the time of the recursive calls, to ensure the termination while proceeding by successive approximations.

Complete lattice

- a lattice iff $\exists \perp \in D$ and $\exists \top \in D$
- complete iff
  - $\forall X \subseteq D \exists U \in D : \forall X \in X x \leq U$ and
  - $\forall X \subseteq D \exists L \in D : \forall X \in X x \geq L$

It is obvious thats $(D, \leq)$ satisfy this conditions.

Monotonicity and continuity Let A be a complete lattice with a partial order $\leq$ and $T : A \longrightarrow A$ a transformation

- T is **monotonous** iff $\forall X, y \in a : X \leq y \Rightarrow T(x) \leq T(y)$
- T is **continuous** iff $\forall X \subseteq a : T(lub(X)) = lub(T(X))$

The transformation that we consider is a functional from a set of function in i ts self.

$$T : (D_a \longrightarrow D_a) \longrightarrow (D_a \longrightarrow D_a)$$

$(TF_a91)([s..t]) = [max(91, s-10)..(t-10)] \cup F_a91(F_a91([(s+11)..min(t+11, 111)]))$

Lemma: T is continuous and monotonous:

- $\forall I_i, I_j \in D_a : I_i \leq I_j \Rightarrow f(I_i) \leq f(I_j)$
- $\forall I_1 \subseteq I_2 \subseteq ... \subseteq I_n \subseteq ... \Rightarrow f(\cup_{i=1...\infty} I_i) = \cup_{i=1...\infty} f(I_i)$

Theorem Let $f([s..t])$, the computing fixpoint consist of T to $f([s..t])$.

- If the constraints on the domain and the operators are satisfied:
  then any fixpoint of T is a correct approximation of the function f
- the smallest fixpoint of T exists and constitutes the best approximation of f
- the smallest fixpoint of T coincide with the limit of an increasing
- the smallest fixpoint of T coincide with the limit of an increasing sequence of approximation: $f_0 \leq f_1 \leq f_2 \leq f_0 ... \leq f_n \leq ...$
  such as: $f_0(I) = \perp \forall I \in D_a$
  $f_{k+1} = T(f_k >) \forall k \geq 0$

2.4.2 Fixpoint Approach

Fixpoint Approach is based on the monotonicity (continuity) of the transformation of the tuples set representing the `pre` and `post` condition for all predicate. Termination of the algorithm in the case of an infinite abstract domain, it did not guarantee. Which is the case if one makes an infinity different recursive call. One can limit oneself to abstract fields finished in certain cases that can be restrive or unacceptable. A possible solution, would be to replace an infinite sequence of approximation by a number of the approximate values.

Approach Widening/Narrowing Suppose the abstract semantics of the program given by a function $f_P : D_{abs} \longrightarrow D_{abs}$. The analysis proceeds as follows:

1. *Widening*: calculation of sequences limit X built by:

$$x_0 = \perp$$
$$x_{i+1} = x_i \text{ and } f_P(x_i)) \sqsubseteq x_i \text{ else} x_i \bigtriangledown f_P(x_i)$$

- *Narrowing* to improve the result obtained by the widening: by

calculating the sequences limit of Y built by:

$$y_0 = \sqcup X$$
$$y_{i+1} = \text{and} f_P(y_i) = y_i \text{then} y_i \text{ else } y_i \bigtriangleup f_P(y_i))$$

Properties

1. *Widening*: $\bigtriangledown : l \times L \longrightarrow L \quad \forall X, Y \in L : \quad X \sqsubseteq Y \bigtriangledown Y$ and $Y \sqsubseteq X \bigtriangledown Y$
   $\rightarrow X \sqcup Y \subseteq X \bigtriangledown Y$
   $\perp \bigtriangledown X = X \bigtriangledown \perp = X$
2. *Narrowing* $\bigtriangleup : l \times L \longrightarrow L$
   $\forall X, Y \in L : \quad Y \sqsubseteq X \Rightarrow Y \subseteq X \bigtriangleup Y \sqsubseteq X$

Widening applied to the intervals

$$[l_0, u_0] \bigtriangledown [l_1, u_1] = \quad [\text{ and } l_1 u_0 \text{then} + \infty \text{else} u_0]$$

Example instead of making the recursive call with $F_a91([-\infty..111])$ one will do it with $F_a91([-\infty.. + \infty])$. But a loss of precision would be introduced. This will allow to speed up the computation of the fixpoint.

3 Refinement

Motivation The abstract interpretation framework establishes a methodology based on rigorous semantics for constructing abstraction that overapproximate the behavior of the program, so that every behavior in the program is covered by a corresponding abstract execution. Thus, the abstract behavior can be exhautively checked for an invatiant in temporal logic. Refinement guided by counterexample consist on approximation of the set of sates that lie on a path from initial state to a bad state which is successsively refine that is done by forward or backward passes. This process is repeated until the fixpoint is reached. If the the resulting set of state is empty then the property is proven. Otherwise, yhe methode does not guaranties that the contreexample trace is genuine.

3.1 Preliminaries

Definition 3.1

Theorem 3.1 `Cousot77`
Let $S = (Q, Q_{init}, \sum, \rightarrow)$ a system representing the semantics of program. The system $S^A = (Q^A, Q^A_{init}, \sum, \rightarrow^A)$ is an abstraction of $S \iff$ there exists a Galois connexion:
$\alpha : \mathcal{P}(Q) \longmapsto \mathcal{P}(Q^A), \gamma : \mathcal{P}(Q^A) \longmapsto \mathcal{P}(Q)$
such that

- $Q_{init} \subseteq \gamma(Q^A_{init})$
- $\forall \tau \in \sum, \forall Q^A_i \subseteq Q^A . post[\stackrel{\tau}{\rightarrow}](\gamma(Q^A_i)) \subseteq \gamma(post[\stackrel{\tau^A}{\rightarrow}](\gamma(Q^A_i)))$

Definition 3.2 Predicat Abstraction `Graf&Saidi97`

Abstract State *Let $Prog = (\mathcal{V}, \mathcal{T} = \{\tau_1, \ldots, \tau_n\}, Init)$ and $\varphi_1, \ldots, \varphi_k$ predicates over the $Prog$'s variables*
we define an abstraction $S^A = (Q^A, Q^A_{init}, \sum, \rightarrow^A)$ as following:

- *$Q^A = B^k$, Q^A is the valuations' set of k boolean variables,*
  any subset Q^A can be represented by a boolean expression over the variables $B_1, \ldots, B_k$
- *S^A as the form $Prog^A = (\mathcal{V}^A, \mathcal{T}^A = \{\tau^A_1, \ldots, \tau^A_n\}, Init^A)$*

Abstract transition *Let $\mathcal{T}^A$, be an abstract transition, it must satisfy the condition of the definition of abstract program , `s.t.` all transition τ, $post[\tau^A](P_B)$, where τ^A is the abstract transition corresponding to τ, have to represent all concrete states q' which are successors by τ of concrete state q represented by P_B. We must show :: $post[\tau](\gamma(P_B)) \Rightarrow \gamma(P'_B)$,*

3.2 Algorithmic checking of refining

This model-checking needs methodological and correctness conditions:

3.2.1 Methodological conditions

- New actions and variables will be introduced by refining
- the variables of refine system and abstract system must be linked by a "collage" invariant.
-

3.2.2 Correctness conditions:

- simulation of the refine system by the abstract
- no cycle between the new action
- no new deadlock

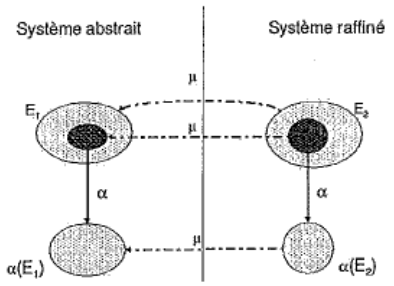


Figure 1. Simulation with old actions

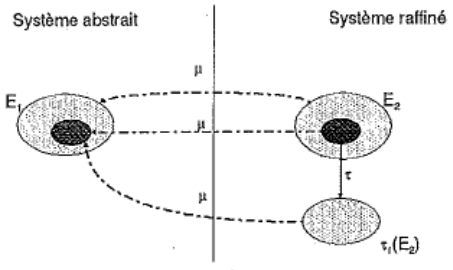


Figure 2. Simulation with new actions

It is a question of carrying out an iteratif calculation of the simulation of TS_2 by TS_1 cfr figures 1 and 2 where transisition α is replace by τ. The algorithm terminates when the fixpoint is reached.

Theorem 3.2

- *If P_1 is property satisfied by TS_1 and TS_2 refines TS_1 then*

$$\frac{TS_1 \models P_1, \vdash TS_2 \subseteq TS_1}{TS_2 \models P_1}$$

- *If P_1 is property satisfied by TS_1 and TS_2 refines TS_1 and if P_2 is a reformulation of P_1 then*

$$\frac{TS_1 \models P_1, \vdash TS_2 \subseteq TS_1}{TS_2 \models P_2}$$

Definition 3.3 *: Let K, K' two systems (resp concrete and abstract). we call false-counterexample or* negative-false *a false universal property in K' but true in K We say that the counterexample specified in K' cannot be reproduced in K*

Corollary If K' is too small, it is very probable that it appears the negative one. If K' is too large, then the checking is not possible the refinement guided by counterexample is thus a natural approach to solve this problem by using a adaptive algorithm which gradually creates an abstraction function by the analysis of false-negative:

Pseudo Algorithm

1. **Initialization:**
   generate a first abstraction function;
2. **Model-Checking:**
   check the model.
   if the checking is a success:
   then
   the specification is correct and
   the algorithm terminates
   else
   generate a counterexample from the abstract model
   verify if this counterexample is a negative-false
   if It is a success then terminate
   else refine the abstract function such that
   the negative-false can be avoid
   goto step 2.

4. Summary

It is thus a question of starting by carrying out an approximation of a way which carries out initial state in a bad condition. Then, a refinement "forwards" or "backward" is carried out, and this process is to repeat until a fixpoint is met. If the resulting set of states is empty then the property is prove, since one no bad condition is reachable, else, nothing guaranteed the value of against example which perhaps distorted by approximation coarse. Heuristics is employed to determine the subset of the reachable states since the initial states. If an equivalence is found, it really acts of an error which can be deferred like a bug, one speaks about positive-false. Abstraction by Predicate : the checking of program by abstraction of closed predicate is a technique of checking of program by abstract interpretation where the abstract domain is composed of the set of guard relating to the states and the transitions from the system. This domain can be generated automatically and checked by a theorems-prover. Like, the set of predicates is always finished, it can be coded by a vector of Boolean, which makes it possible on the other hand to use the model-checker for calculations of fixpoint.Si, the domain is very large, one can use a chaotic iterator and to use a widening if it is necessary of speed up the convergence. The termination and reachability decidable in this case. The one limitation of this technique of checking by predicate abstraction is that the processes of refinement, which primarily consists in calculating the weakest invariant, are extremely slow. This obligates the users to require at least the atomic predicate necessary to the proof. This fact the human intervention which specific is given must be repeated for different programs even if they are very similarities. and it

5. Conclusion and Future Work

It has been shown that static checker can cover a large number of potential faults, their automatic usage is still far from realistic. However, as a verification step prior to testing or code review, static checkers, can already enhance the software development process today. Several techniques like Altarica, B or CSP2B were proposed to specify and check reactive systems by using hierarchic development by refinement. In this case, the systems design is realized gradually by increasing the systems design to each step of the specification from a very abstract sight of the system until its implementation. For us, a system implements (refines) another system if all the traces of execution of the most detailed system are too traces of the most abstract (modulo the introduction of details during refinement). The checking of the system thus will use refinement to model the initial system in a more precise way, if the model-checker provides a erroneous result consequence of coarse approximation at the time of the abstraction.

References

[1] **Edmound Clarke** Contreexemple-guided abstraction refinement. In 10Th International Symposium on Temporal Representation and Reasonning and Fourt International Confernce on Temporal Logic, 2003.

[2] **P. Cousot, R. Cousot** *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints*, POPL 1977, Sigact Sigplan, pp 238–252.

[3] **T. Henzinger, R.Jhala, R.Majumdar, G. Sutre** *Lazy abstraction* ; Technical report University of California and LaBRI, 2002.

[4] **T. Ball, A. Podelski, S. Rajammani** *Boolean and cartesian abstractionfor model checking C programs.* ; Technical report LORIA,Microsoft Corporation, 2000.

[5] **Hassen Sadi** *Model Checking Guided Abstraction and Analysis*. SAS 2000: 377-396