Software Verification Techniques based on Floyd's Method

Thomas Parsch
parsch@in.tum.de

***Abstract*—This report is about software verification and gives a brief introduction with examples to Floyd's method, lazy abstraction and the abstraction with the help of Craig interpolation.**

I. Introduction

One of the most important foundations of software verification is Floyd's method [3] from 1967. Its induction like approach and the association of program statements with logical arguments are described in section II. Two years later Floyd's ideas were picked up by Hoare in order to postulate his Hoare logic[6].

Although it was possible to verify little programs, the computation effort needed for larger code samples exceeded the capabilities by far. The amount of predicates to keep track of was too large. In the years afterwards new techniques were developed in order to reduce it. One of two solutions, which are presented in this paper, is called lazy abstraction [4]. This method allows to maintain a preciser model than before, because predicates, which should be tracked of, are discovered in an automatic way instead of tracking everything.

If a predicate is put on the list for a branch of the verification tree, it is not removed until a leaf is reached, even if it is not needed anymore. The second technique [5] makes use of Craig interpolation in order to have always a parsimonious set of predicates being checked.

In the following the Floyd's method, lazy abstraction and abstraction with the help of Craig interpolation will be explained, based on the mentioned papers.

II. Floyd's method

In order to proof correctness, equivalence and termination each statement of a program is associated with a verification condition $V_c(P, Q)$. The verification condition holds for a statement S if the propositions P, also called antecedent and Q, often called consequent, hold. P is evaluated directly before the statement is executed and Q after the statement. If there is branching (e.g. if statements) and joining in the program , P could consist of several entrance propositions $\{P_1, P_2, P_3, ...\}$ and Q of several exit propositions $\{Q_1, Q_2, Q_3, ...\}$. The selected entrance and the chosen exit propositions have to hold.

```
1 // int i is set to 5
2 int x =0;
3 while(i>0){
4   x++;
5   i--;
6 }
7 return x;
```

Listing 1. Simple example program

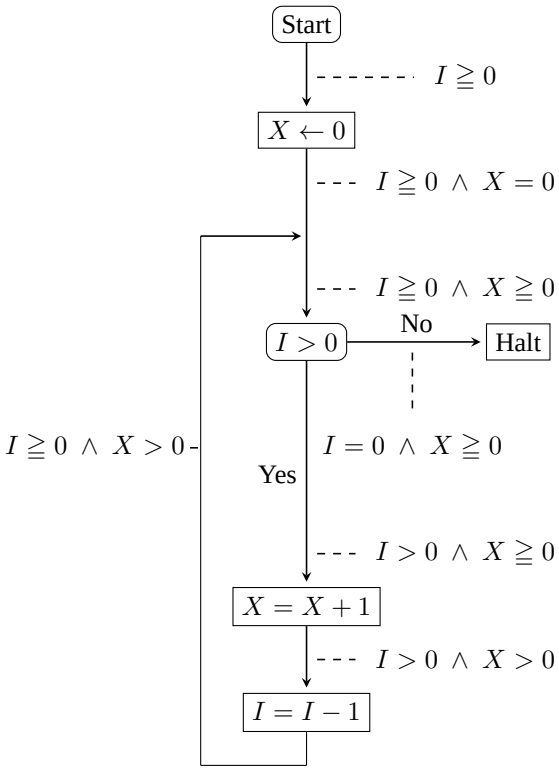

Fig. 1. Floyd's method control chart example

$\{P\}S\{Q\}$ is also called Hoare triple. [6] The consequent of a statement should be the antecedent of the next statement, this has similarities to an induction, where the next proof step is based on the step before. It is possible to visualize a program as a flowchart, where the vertexes represent the statements and the edges are attached with a tag, which include the antecedent respectively consequent. An example for the code in listing 1 is given in figure 1.

In the example it is assumed that the variable *int i* is already set to a value greater than 0. So the first tag is $I \geqq 0$. In the statement 0 is assigned to x. So we know that the variable X, which represents *int x* is equal to 0. Since at the moment the proposition of the loop is not known yet, the next statement $I > 0$ is taken directly into account. Because of the *if*-statement there are two possible exits. If $I > 0$ then $I > 0 \wedge X = 0$ has to hold, otherwise $I = 0 \wedge X = 0$. In the latter case the program would terminate. Following the other branch, after the statement $X = X + 1$ the proposition is $I > 0 \wedge X > 0$. And so before the loop goes back in the chart $I = I - 1$ is executed and $I \geqq 0 \wedge X > 0$ has to hold.

Since $I \geqq 0 \wedge X > 0$ and $I = 0 \wedge X = 0$ does not exclude each other it is possible to combine it to $I \geqq 0 \wedge X \geqq 0$. If you follow the statements further the propositions as seen on the figure 1 are received.

The combination and more conversions are allowed because [3] postulates following axiomes.

Axiom 1:
If $V_c(\mathbf{P};\mathbf{Q})$ and $V_c(\mathbf{P'};\mathbf{Q'})$, then $V_c(\mathbf{P} \wedge \mathbf{P'};\mathbf{Q} \wedge \mathbf{Q'})$

If more than one proposition holds for an antecedent or consequent, this axiom allows to combine them.

Axiom 2:
If $V_c(\mathbf{P};\mathbf{Q})$ and $V_c(\mathbf{P'};\mathbf{Q'})$, then $V_c(\mathbf{P} \vee \mathbf{P'};\mathbf{Q} \vee \mathbf{Q'})$

This allows the combination of the results of a case analysis. This is for example useful, when several ranges of initial values are checked for certain variables.

Axiom 3:
If $V_c(\mathbf{P};\mathbf{Q})$, then $V_c((\exists x)(\mathbf{P});(\exists x)(\mathbf{Q}))$

This axiom is very fundamental and allows the sequential combination of antecedents and consequents.

Axiom 4:
If $V_c(\mathbf{P};\mathbf{Q})$ and $\mathbf{R} \vdash \mathbf{P}$, $\mathbf{Q} \vdash \mathbf{S}$, then $V_c(\mathbf{R};\mathbf{S})$

Corollary:
If $V_c(\mathbf{P};\mathbf{Q})$ and $\vdash (\mathbf{P} \equiv \mathbf{R})$, $\vdash (\mathbf{Q} \equiv \mathbf{S})$, then $V_c(\mathbf{R};\mathbf{S})$

"Axiom 4 asserts that if P and Q are verifiable as antecedent and consequent for a command, then so are any stronger antecedent and weaker consequent." [3]

The axioms can be also deduced from the properties of completeness and consistency.

Applying this method to a programming language a semantic definition is needed. First a syntactic definition must be defined, to check the syntactically correctness and to identify the basic structure of the language. In addition for every possible statement, a rule, how to construct the verification condition, has to be made. The semantic definition must be consistent and should be complete. Once this is done, a program can be verified.

It is not always possible to proof the termination of a program, since the halting problem is unsolvable in general. Though for programs for which a W-function could be defined, it is possible. A W-function calculates from the free variables of a proposition and the program counter a value, which belongs to a well-ordered set. If the value gets smaller, with every step executed, compared to the one before, the program terminates.

III. Lazy Abstraction

In order not to keep track of all predicates as described in chapter II, the lazy abstraction[4] technique works after the abstract-check-refine paradigm [1].

Abstraction
The abstraction step uses a finite set of predicates to create an abstract model of the program.

Verification
The following verification step checks the abstraction. If no counter example is found the program is correct. Otherwise an abstract counterexample is returned.

Counterexample-driven refinement
If an abstract counterexample is found, there are two possibilities. Either it is possible to relate it to a counterexample in the program. So the verification stops with the result that the program is incorrect. Or it is a spurious counterexample, which means that the abstraction is too coarse. Then the abstraction model has to be refined by adding more predicates to the finite set. If this is done the cycle is repeated, starting with the abstraction step.

```
1  Example() {
2    if(*){
3      do {
4        got_lock = 0;
5        if (*) {
6        lock();
7        got_lock++;
8        }
9      } while (*)
10   }
11   do {
12     lock();
13     old = new;
14     if (*) {
15       unlock();
16       new++;
17     }
18   } while (new != old);
19   unlock();
20   return();
21 }
```

Listing 2. Lazy abstraction: Code example from [4]

```
22 lock(){
23   if (LOCK == 0) {
24     LOCK = 1;
25   } else {
26     ERROR
27   }
28 }
```

Listing 3. Lazy abstraction: lock() function from [4]

```
30 unlock() {
31   if (LOCK == 1){
32     LOCK = 0;
33   } else {
34     ERROR
35   }
36 }
```

Listing 4. Lazy abstraction: unlock() function from [4]

To realize the cycle mentioned above, a control flow automaton (CFA) is created from the code. The different program states are connected by edges. These are labeled with either a block of instructions, executed to reach the next state, or an assumption about predicates (e.g. if conditionals). The CFA for listing 1 can be seen in figure 2. The $*$ represents further statements, which are not modeled in this example. So all emanating branches could be taken and the corresponding predicate assumed as $true$. The examples of this section are taken from [4].

In order to run through the three highlighted phases, lazy abstraction performs a forward search for the verification and a backwards counterexample analysis. The following describes this two steps accompanied by an example.

Forward search
Based on the CFA a search tree is created. First a depth-first search is performed in order to compute the reachable regions. The predicates to consider are stored in a set. The set could be empty in the beginning or filled with some predefined predicates. An assumption about a predicate which is not in the current set is assumed to be true. Instructions including unknown predicates are simply ignored. With this method a reachable region for every node in the search tree is created, until an error state is reached or the search tree covers all possible states. If a loop in the CFA is followed and the examined state is a state, which was visited before,

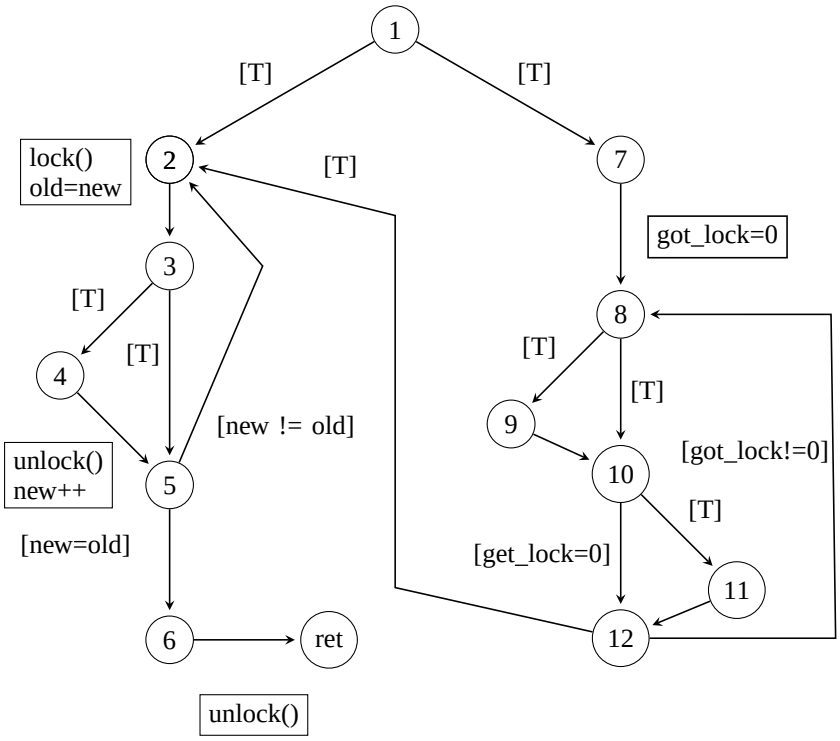

Fig. 2. Lazy abstraction: Control Flow Automaton (CFA) from [4]

two different cases must be taken into consideration. If no new states are added to the union of the reachable regions, the search goes back to the next branch. Otherwise the branch of the search tree is followed further.

In the example figure 3 it is assumed that the set already includes the predicate $LOCK = 0$ and $LOCK = 1$. In addition in the beginning $LOCK = 0$ holds. The predicates without curly brackets are from the forward search. The remaining from the following backwards counterexample analysis. The forward search is started from the root node 1. From node 1 to 2 nothing changes, since the step is not modeled and nothing is executed. But in the next step the formula describing the reachable region, changes. $old = new$ is not taken into account, since it is not in the set of tracked predicates. The other statement $lock()$ in contrast changes the predicate of the reachable region to $LOCK = 1$. This is a simplification, since the $lock()$ function could be also represented with several nodes in the CFA. From node 3 to 4 nothing happens, since it is not modeled. In the following step, the reachable region is described with $LOCK = 0$, because of the $unlock()$ function. $new++$ is not taken into account. The step from node 5 to 6 do not change anything, since $new = old$ is not included in the set of predicates. Finally an error state is reached, because $LOCK = 0$. Beside the $ERROR$ statement nothing is executed, the reachable region could be described by $LOCK = 0$.

Backwards counterexample analysis

If an error state, as in the example above, is discovered, it must be checked if it is a real error or a spurious counterexample. Is the latter the case, new predicates have to be added to the set in order do refine the abstraction. Therefore a backtrack analysis from the error node towards the root is executed. It starts on the error node and computes for every predecessor of the node the bad region with respect to the labels of the bad path. If the intersection of the bad region and the reachable region from the forward search is empty this node is called pivot node. This should be the case since the error should not be reachable from the root node. But if no pivot node is found the counterexample belongs to a real error. Otherwise the set of predicates considered is expanded from the pivot node on with the predicates derived from the bad region and the forward search starts again. The advantage of this approach that not all predicates have to be tracked and in addition different subtrees could use different sets of predicates. By keeping subtrees computation time could be saved.

As alread mentioned and the name reveals, backwards counterexample analysis starts from the error and goes backwards to the root. Thereby an formula describing the bad region is constructed. The first bad region to be deduced is the one of node number 6. The weakest precondition to reach from this node the error node is $LOCK = 0$. Since only if $LOCK = 0$ an error occurs when $unlock()$ is called. In order to get from node 5 to 6 $new = old$ is assumed, so $LOCK = 0$ & $new = old$ is the related bad region. In the following step from 4 to 5 $unlock()$ and $new++$ is executed. Whereat $new++$ can be seen as $new = new + 1$. So the weakest precondition is $LOCK = 1$ & $new + 1 = old$. Still the intersection between the bad region and reachable region is not empty. From node 3 to 4 nothing happens, so we can carry over the bad region. In order to get from node 2 to 3 $lock()$ and $old = new$ have to be executed. So in the corresponding bad region $LOCK = 0$, because there must be no Lock, when executing $lock()$ without an error and $new + 1 = new$ if $new + 1 = old$ should be true after executing $old = new$. Since the bad region itself is empty, because $new + 1 = new$

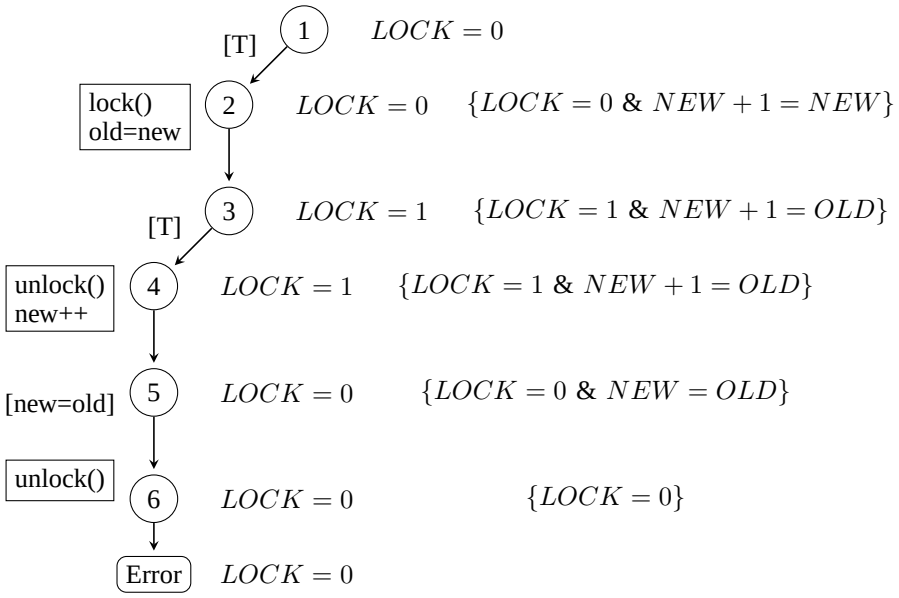

Fig. 3. Lazy abstraction: Forward search and backward counterexample analysis similar to [4]

has no solution, the intersection between the reachable region and the bad region is empty. Consequently the node must be the pivot node. As node 2 is inside the CFA, the counter-example is a spurious one and $new = old$ can be added to the set of tracked predicates from the pivot node on.

Now a new forward search starting from the root node can be done as seen in fig 4. This time a return statement in the left branches of the search tree is reached, because the error states are not reached anymore, because the relevant predicates are taken into account. Node 2 in the lower left is assumed as leaf, since the states reached with $LOCK = 0$ & $!new = old$ are completely included in the ones reachable with $LOCK = 0$. Two other branches are not followed, because the assumption does not hold. This is marked by a crossed line.

Still a new spurious counter-example would be found in the right side of the tree and the predicate $got\_lock = 0$ has to be added for the pivot node number 7. Due to that fact for the whole tree $LOCK = 0$ & $LOCK = 1$ has to be taken into account and $new = old$ starting from pivot node 2 and $got\_lock = 0$ beginning from pivot node 7. This can be seen in figure 5.

IV. Abstraction with Craig interpolation

The following concept is found in [5]. Relationships which are only specified between current values of variables and those, which are required for proving correctness are called parsimonious. To be able to calculate always a parsimonious set of predicates the Craig interpolation [2] is used. A Craig interpolation $\psi = Craig(\varphi^-, \varphi^+)$ of two formulas φ^-, φ^+ fulfills the following properties:

- φ^- implies ψ
- $\varphi^- \wedge \psi$ is unsatisfiable
- ψ only contains symbols from φ^- and φ^+

Like the lazy abstraction approach, abstraction with Craig interpolation is also counterexample driven. But unlike above

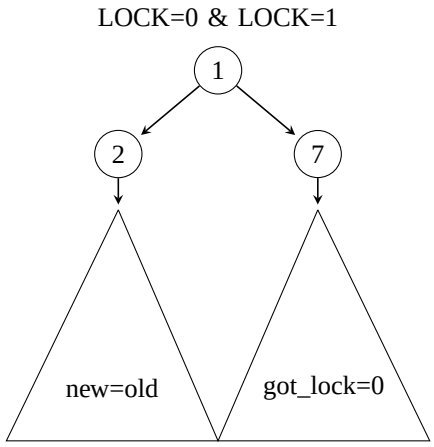

Fig. 5. Different abstractions similar to [4]

a predicate is only tracked, when it is really needed and not all the way downwards from the pivot node on. Therefore - starting from an error trace - for every node the Craig interpolation is calculated. In listing 6 we seen an error trace, for the same case as in figure 3.

```
assume( lock = 0 )
lock = 1
old = new
assume (lock = 1)
lock = 0
new = new +1
assume (new = old)
assume (lock = 0 )
Error
```

Listing 5. Error trace

From this the constraints are derived. As values change during the program, it is important to introduce the subindices, so that a value of a variable in the beginning of the program does not lead to inconsistency, although it already changed its

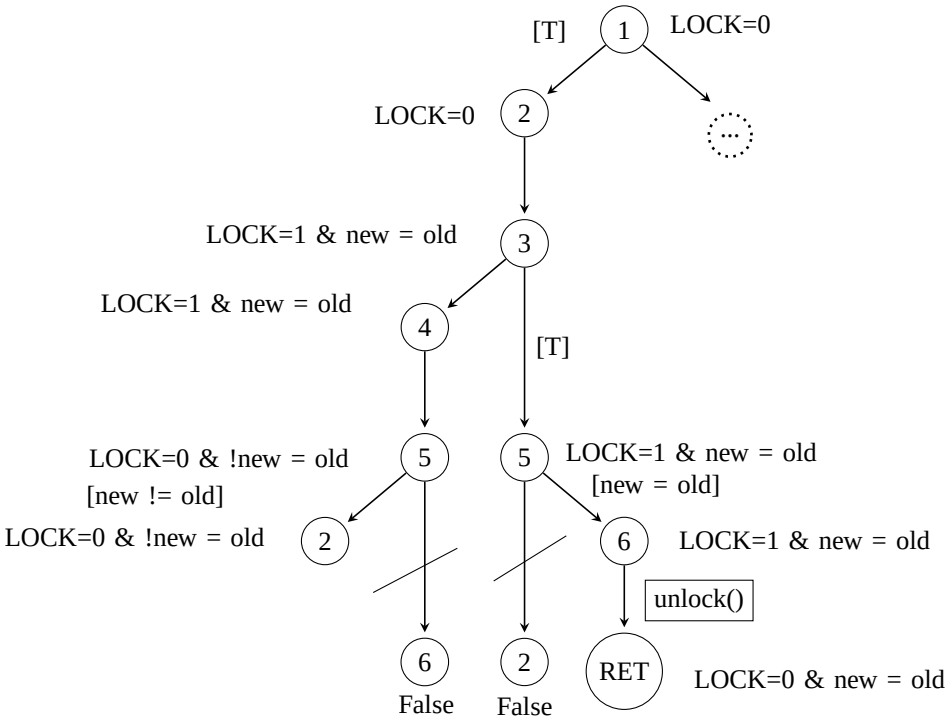

Fig. 4. Search with new predicate similar to [4]

values. The derived constraints from the error trace in listing 5 can be seen in listing 6.

$L_0 = 0$
$L_1 = 1$
$old_0 = new_0$
$L_1 = 1$ discard
$L_2 = 0$
$new_1 = new_0 + 1$
$new_1 = old_0$
$L_2 = 0$ discard

Listing 6. Constraints

Some formulas can be discarded once, because they are redundant. For every node (beside the error node) a Craig interpolation has to be calculated. Therefore φ_1^- is the concatenation of formulas from root to (including) the one related to the actual node. φ_1^+ is the concatenation of the remaining formulas. From a proof of unsatisfiability of $\varphi_1^- \wedge \varphi_1^+$ the interpolation could be derived. Therefore only constraints have to be taken into account, where an included symbol appears on both sides. In the following the interpolant ψ is calculated for every node of the example.

Here the trick is used that instead of φ_n^- it is possible to take $\psi_{n-1} \wedge constr_{\text{new}}$. Whereas $constr_{new}$ is the constraint that switched from φ^+ to φ^-.

φ_1^-: L_0
φ_1^+: $L_1 \wedge old_0 = new_0 \wedge L_2 = 0$
$\wedge\, new_1 = new_0 + 1 \wedge new_1 = old_0$
ψ_1: $true$

φ_2^-: $true \wedge L_1 = 1$
φ_2^+: $old_0 = new_0 \wedge L_2 = 0 \wedge new_1 = new_0 + 1 \wedge new_1 = old_0$
ψ_1: $true$

φ_3^-: $true \wedge old_0 = new_0$
φ_3^+: $L_2 = 0 \wedge new_1 = new_0 + 1 \wedge new_1 = old_0$
ψ_3: $old_0 = new_0$

φ_3^-: $true \wedge old_0 = new_0$
φ_3^+: $L_2 = 0 \wedge new_1 = new_0 + 1 \wedge new_1 = old_0$
ψ_3: $old_0 = new_0$

φ_4^-: $old_0 = new_0 \wedge L_2 = 0$
φ_4^+: $new_1 = new_0 + 1 \wedge new_1 = old_0$
ψ_4: $old_0 = new_0$

φ_5^-: $old_0 = new_0 \wedge new_1 = new_0 + 1$
φ_5^+: $new_1 = old_0$
ψ_5: $old_0 = new_1 - 1$

φ_6^-: $old_0 = new_1 - 1 \wedge new_1 = old_0$
φ_6^-: $\varnothing$

The corresponing predicate, which should be kept track of, could be simply found, by removing the subindices from the formula. So e.g. for ψ_4: $old_0 = new_0$ it is $old = new$. This technique improves the perfomance of the verification process clearly.

REFERENCES

[1] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *Computer Aided Verification*, pages 154–169, 2000.

[2] W. Craig. Linear reasoning. *Journal of Symbolic Logic*, 22:250–268, 1957.

[3] R. Floyd. Assigning meanings to programs. *Proc. Symp. on Applied Mathematics*, 19:19–32, 1967.

[4] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. *POPL*, 2002.

[5] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstraction from proofs. *POPL*, 2004.

[6] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 1969.