SMT-based Test-Case Generation with Complex Preconditions

Ricardo Peña Jaime Sánchez-Hernández Miguel Garrido Javier Sagredo
Complutense University of Madrid, Spain*
{ricardo,jaime}@sip.ucm.es,jasataco@gmail.com,migarr01@ucm.es

We present a system which can automatically generate an exhaustive set of black-box test-cases, up to a given size, for programs under test requiring complex preconditions. The key of the approach is to translate a formal precondition into a set of constraints belonging to the decidable logics of SMT solvers. By checking the satisfiability of the constraints, then the models returned by the solver automatically synthesize the cases.

We also show how to use SMT solvers to automatically check for validity the test-case results, and also to complement the black-box cases with white-box ones. Finally, we use of solver to perform what we call automatic partial verification of the program. In summary, we propose a system in which exhaustive black-box and white-box testing, result checking, and partial verification, can all be done automatically. The only extra effort required from programmers is to write formal specifications.

1 Introduction

Testing is very important for increasing program reliability. Thorough testing ideally would exercise all the different situations described in the specification, and all the instructions and conditions of the program under test, so that it would have a high probability of finding bugs, if they are present in the code. Unfortunately, thorough testing is a time consuming activity and, as a consequence, less testing than the desirable one is performed in practice.

There is a general agreement that automatic tools can alleviate most of the tedious and error prone activities related to testing. One of them is test-case generation (TCG). Traditionally (see, for instance [1]), there are two TCG variants: black-box TCG and white-box TCG. In the first one, test-cases are based on the program specification, and in the second one, they are based on a particular reference implementation. Each one complements each other, so both are needed if we aim at performing thorough testing.

On white-box testing there is much literature, and sophisticated techniques such as symbolic and concolic TCG, have been proposed. One of the problems arising when using the implementation as the basis to generate cases, is that less attention is paid to program specification, that usually is even not formalized. As a consequence, test cases can be generated that do not meet the input requirements expected by the program. Generating cases satisfying a given precondition may not be so easy when that precondition is complex. For instance, some programs require as input sorted arrays, sorted lists, or sophisticated data structures satisfying complex invariants such as red-black trees or binary heaps. When this happens, usually auxiliary boilerplate code is created to build this structures in order to generate valid input for the program under test. This approach increments the testing effort and may also introduce new errors in the boilerplate code.

*Work partially funded by the Spanish Ministry of Economy and Competitiveness, under the grant TIN2017-86217-R, and by the Madrid Regional Government, under the grant S2018/TCS-4339, co-funded by the European Union EIE funds.

Submitted to:
PROLE 2019

Another problem related to testing automation is checking for validity the results returned by the program for each test-case. If this checking is done by humans, then again the testing effort and the possibility of errors increase. In order to automate this process, programmers must write formal post-conditions, or formal assertions, and have executable versions of them, as it is done for instance in property-based testing [7, 5]. Again, having executable assertions require programming effort and open more possibilities of introducing errors. Our group recently presented a tool [2] that transformed assertions written in a logic language supporting sets, multisets, sequences and arrays, into executable Haskell code, so that no extra code was needed in order to create executable versions of program postconditions. This system could also be used to generate cases satisfying a complex precondition. The approach consisted of automatically generating a huge number of structures of the appropriate type, then executing the precondition on each of them, and then filtering out those satisfying it. For instance, if a precondition required a tree to be an AVL one, then all possible trees up to a given size were generated and then only those satisfying the executable *isAVL* predicate were kept. The approach was fully automatic, but not very efficient, as sometimes less then 1% of the generated cases were kept.

In this work, we use an SMT solver [3] to directly generate test-cases satisfying a complex precondition, without needing a trial and error process, such as the above described one. The key idea is to transform the formal precondition into a set of constraints belonging to the logics supported by the SMT solver. This has been possible thanks to a recent facility introduced in such solvers: the theory of algebraic types and the support given to recursive function definitions on those types. If the constraints are satisfiable, then every model returned by the SMT solver is a test-case satisfying the precondition. In this way, we have been able to generate sorted arrays, sorted, lists, AVL trees, red-black trees, and some other complex data structures. We apply the same approach to the postconditions: the result returned by the program under test is converted into an SMT term, and then the postcondition constraints are applied to it. If the resulting SMT problem is satisfiable, then the result is correct.

In many programs, the conditions occurring in **if** statements and **while** loops are simple enough to be converted into constraints supported by the SMT logics. If this is the case, then we can use SMT solvers to also generate white-box test-cases, as it is done in the symbolic testing approach. The execution paths along the program are converted into appropriate constraint sets, and then they are given to the solver. In this work, we add to these path constraints the precondition constraints, so we generate test-cases which are guaranteed to satisfy the precondition, and also to execute a given path.

A final contribution of this work is to use the SMT solver to do 'effortless' program verification. We call the approach *partial automatic verification* and consists of giving the solver formulas to be checked for validity instead of for satisfiability. Each formula expresses that all test-cases satisfying the precondition, satisfying the constraints of a given path, and returning some result in that path, must also satisfy the postcondition for the returned result. If the solver checks the validity of this formula, this is equivalent to proving that *all* valid inputs exercising that path are correct. If we apply this procedure to an exhaustive set of paths covering all the program under test up to a given depth, we would be verifying the correctness of a high number of test-cases at once, without even executing the program.

Summarizing, we propose a system in which exhaustive black-box and white-box testing, result checking, and partial verification, can all be done automatically. The only extra effort to be done by programmers is writing formal specifications for their programs.

The plan of the paper is as follows: in Sec. 2, we explain how to program complex preconditions in the SMT language; Sec. 3 shows how to generate exhaustive black-box test-cases by only using the precondition; Sec. 4 explains the generation of exhaustive white-box test-cases, and Sec. 5 details the partial verification approach. Finally, Sec. 6 surveys some related and future work, and draws some conclusions.

2 Specifying Pre- and Postconditions by using the SMT Language

In the last fifteen years, SMT solvers have evolved very quickly. There has been a very much profitable feedback between SMT developers and SMT users. The latter have been asking for more and more complex verification conditions to be automatically proved by SMTs, and the former have improved the heuristics and search strategies of SMTs in order to meet these requirements. The result has been that SMT solvers support today a rich set of theories, and that more complex programs can be verified by using them.

Up to 2015, the available SMTs essentially supported the boolean theory, several numeric theories, and the theory of uninterpreted functions. With the addition of a few axioms, the latter one was easily extended in order to support arrays, set and multiset theories. This state of the art is reflected in the language described in the SMT-LIB Standard, Version 2.5.[1]

After that, in the last few years, SMT solvers have incorporated algebraic datatypes and the possibility of defining recursive functions on them. Reynolds et al [13] proposed a strategy for translating terminating recursive functions into a set of universally quantified formulas. In turn, several techniques have been developed to find models for a large set of universally quantified formulas. From the user point of view, a satisfiability problem can be posed to a modern SMT solver in terms of a set of recursive function definitions. The SMT-LIB Standard, Version 2.6, of Dec. 2017, reflects this state of affairs.

It happens that many useful preconditions and postconditions of algebraic datatype manipulating programs can be expressed by means of recursively defined predicates and functions. For instance, the function that inserts an element into an AVL tree, needs as a precondition the input tree to be an AVL. An AVL is defined as either the empty tree, or as a non-empty tree whose two children are both AVL trees. Additionally, the tree must be a Binary Search Tree (i.e. a BST predicate holds), and the difference in height of the children should be at most one. Also, in each node there is an additional field containing the tree height. In turn, the BST property can be recursively defined, and so can it be the height function on trees. In Fig. 1, we show some of these definitions written in the SMT language. There, the predefined function `ite` has the same meaning as the `if-then-else` expression of functional languages. In a similar way, predicates for defining sorted lists, or invariants of data structures such as skew heaps, leftlist heaps, or red-black trees, can be defined.

The same idea applies to postconditions. For instance, after inserting an element in an AVL tree, we would like to express that the result is also an AVL tree, and that the set of elements of this result is the union of the set of elements of the input tree, and the element being inserted. The function giving the set of elements of a tree, or the multiset of elements of a heap, can also be recursively defined in the SMT-LIB language, as we show in Fig. 2. There, a set of elements is modeled as an infinite array of boolean values, i.e. as the characteristic function of the set. Having defined such predicates and functions, the specification of the insertion function for AVLs, can be written as follows:

$$\{isAVL(t)\} \tag{1}$$

$$\textbf{define } insertAVL\,(x :: int, t :: AVL\ int) :: (res :: AVL\ int)$$

$$\{isAVL\,(res) \wedge setA\,(res) = set\text{-}union\,(setA\,(t), unit\,(x))\} \tag{2}$$

[1] `http://smtlib.cs.uiowa.edu/language.shtml`

```
(declare-datatypes (T) ((AVL leafA (nodeA (val T) (alt Int) (izq AVL) (der AVL)))))

(define-fun-rec
   heightA ((t (AVL Int))) Int
     (ite (= t leafA)
          0
          (+ 1 (max (heightA (izq t)) (heightA (der t))))
     )
)

(define-fun-rec
   isAVL ((t (AVL Int))) Bool
     (ite (= t leafA)
          true
          (and (isBSTA t) (isAVL (izq t)) (isAVL (der t))
               (<= (absol (- (heightA (izq t)) (heightA (der t)))) 1)
               (= (alt t) (heightA  t))
          )
     )
)

(define-fun-rec
     isBSTA ((t (AVL Int))) Bool
     (
          ite(= t leafA)
               true
               (ite (and (= (izq t) leafA) (= (der t) leafA))
                    true
                    (ite (= (izq t) leafA)
                         (and (isBSTA (der t)) (< (val t) (minTA (der t))))
                         (ite (= (der t) leafA)
                              (and (isBSTA (izq t)) (< (maxTA (izq t)) (val t)))
                              (and (isBSTA (izq t)) (isBSTA (der t))
                                   (< (maxTA (izq t)) (val t)) (< (val t) (minTA (der t))))
                         )
                    )
               )
     )
)
```

Figure 1: Definition of the predicate `isAVL` and the function `heightA` in the SMT language

```
(define-sort Set (T) (Array T Bool))

(define-fun empty () (Set Int)
  ((as const (Array Int Bool)) false))

(define-fun unit ((i Int)) (Set Int)
  (store empty i true))

(define-fun set-union ((s1 (Set Int)) (s2 (Set Int))) (Set Int)
  ((_ map or) s1 s2 )  )

(define-fun-rec
   setA ((t (AVL Int))) (Set Int)
      (ite (= t leafA)
           empty
           (set-union (set-union (setA (izq t)) (setA (der t))) (unit (val t)))
      )
)
```

Figure 2: Definition of the function `setA` giving the set of elements of an AVL tree

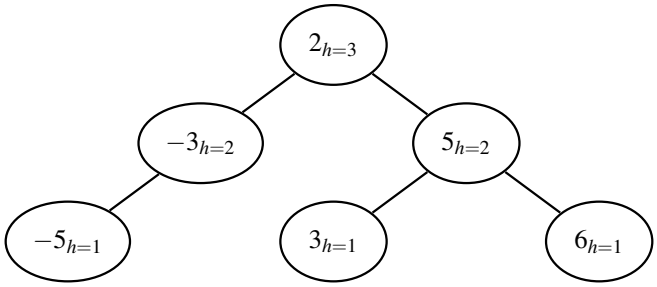

Figure 3: AVL tree synthesized with a size constraint `(assert (= 3 (heightA t)))`.

3 Synthesizing Black-Box Test-Cases

The purpose of defining preconditions as a set of constraints written in the SMT-LIB language is to let solvers to check the satisfiability of these constraints and to generate models for their variables. Each model satisfying the constraints is then easily transformed into a test-case, since the generated values satisfy the precondition.

In our context, the Unit Under Test (UUT in what follows) is a set of related pure functions, of which only one is visible from outside the unit. The functions do not share any variable or object. The only allowed interaction between them is that the visible one may call any of the other functions, and the latter ones may call each other in a mutual recursive way. The visible top-level UUT function is specified by means of a formal precondition and postcondition.

Modern SMT solvers are very clever in finding models for constraints involving algebraic datatypes and recursive functions on them. In our experiments, we used the Z3 SMT solver [12], version 4.8.4, published in January 2019. When giving the following constraint to Z3:

```
(assert (= 3 (heightA t)))
```

it synthesizes a binary tree t having height 3. Previous versions of this solver were not able to process this constraint. They entered an endless loop when trying to solve it. This means that the search strategies of SMT solvers are continuously evolving, and that each time they can solve more complex problems.

Our preconditions are conjunctions of predicates usually involving recursive definitions, as those shown in Fig. 1. Our aim is to generate an exhaustive set of black-box test-cases, so in order to have a finite number of them, the user should fix a maximum size for each argument of the UUT. In the cases of trees and lists, their height and their length would be suitable sizes. In the case of integers, the user should specify a finite interval for them. If tree nodes and list elements consist of integer values, this latter constraint has the additional effect of limiting the values which will populate the nodes, so contributing to getting a finite number of models for the arguments.

In the example shown in Fig. 3, we have forced the tree t to satisfy the predicate $isAVL(t)$ and to have 3 as its height, and the integer values to be comprised between -6 and $+6$. The solver generated the model shown in the drawing, which is just one of all the possible models. The annotation $h = xx$ in each node is the value of its height field.

The next step is to be able to generate many different test cases. The most popular solvers, such as Z3 [12], only return one model for every satisfiable problem. There exist the so called All-SAT solvers, which give as a result *all* the models satisfying a set of constraints (see for instance [14]), but these only solve pure SAT problems, i.e. they lack the rich set of theories supported by SMT solvers. A common

solution to force the SMT to give more models is adding constraints which negate the prior obtained models. This is what we did: we called the solver in a loop, adding at each iteration a constraint forcing the result to be different from the previous one. For instance, for AVL trees having less than or equal to 7 nodes, and populated with values between 1 and 10, we obtained 3353 trees. The solver entered an infinite loop trying to get the model number 3354. In fact, our hand calculations gave us that there exist only 3353 correct AVLs within the above restrictions. This means that the solver was trying to solve an unsatisfiable problem and got lost while searching new models. Getting timeouts in SMT problem solving is a frequent situation when the set of constraint reaches a high complexity, as it is the case here.

Then, summarizing our proposal for generating black-box test-cases, it consists of first fixing a size for the data structure, then fixing an interval of values for populating the elements of the structure, and then letting the solver to find *all* the models satisfying the constraints. This amounts to perform exhaustive testing up to a given size. We consider the set of cases generated in this way to have a high probability of finding bugs, if they are present in the program. Successfully passing these tests will give us a high confidence in the UUT reliability, since it is infrequent for programmers to build programs that successfully pass exhaustive testing for small size test-cases, and still contain bugs, only showing up for bigger sizes.

The last step consists of checking that the result computed by the UUT for each test-case is really the correct one. Given the high number of test-cases we plan to generate, ideally this should be automatically done by the system. The result could be a value of a simple type, such as an integer or a boolean value, or it could be a term of an algebraic type, such as a list or a tree. Our proposal is to translate the returned term from the executable language to the solver language, and then to use the postcondition to check whether the term satisfies it. For instance, if *res* is the tree returned for the function inserting a value x into an AVL tree t, then the SMT solver should perform a satisfiability check for the following constraint (see eqn. 2):

```
(assert (and (isAVL res) (=  (setA res) (set-union (setA t) (unit x)))))
```

where t, x, and *res*, are instantiated variables. If the solvers returns `sat`, then the result is correct. Otherwise, it is not.

4 Synthesizing White-Box Test-Cases

The black-box test-cases are independent of the possible implementations of the UUT. Given a particular implementation, it may be the case that, after running an exhaustive set of black-box test-cases up to a given size, there still exist non exercised paths in the UUT. The usual testing strategies recommend to additionally generate implementation-based test-cases which complement the black-box ones. A common exhaustiveness criterium is to generate cases exercising all the paths in the UUT. When paths through loops are considered, then a bound on the number of iterations through the loop is set. For instance, a bound of 1 means that paths exercising at most one iteration of each loop are considered.

Our implementation language is a sort of core functional language which supports mutually recursive function definitions. It is the internal language (we call it IR, acronym of Intermediate Representation) of our verification platform CAVI-ART [11, 10], to which real-life languages such as Java and Haskell are translated. In Fig. 4 we show its abstract syntax. Notice that all expressions are flattened in the sense that all the arguments in function and constructor applications, and also the **case** discriminants, are atoms. An additional feature is that IR programs are in **let** A-normal form [6], and also in SSA[2] form, i.e. all

[2] Static Single Assignment.

a	::=	c	{ constant }
	|	x	{ variable }
be	::=	a	{ atomic expression }
	|	$f\ \overline{a_i}$	{ function/primitive operator application }
	|	$\langle \overline{a_i} \rangle$	{ tuple construction }
	|	$C\ \overline{a_i}$	{ constructor application }
e	::=	be	{ binding expression }
	|	**let** $\langle \overline{x_i :: \tau_i} \rangle = be$ **in** e	{ sequential let. Left part of the binding can be a tuple }
	|	**letfun** $\overline{def_i}$ **in** e	{ let for mutually recursive function definitions }
	|	**case** a **of** $\overline{alt_i}[;\ \_ \to e]$	{ case distinction with optional default branch }
$tldef$	::=	**define** $\{\psi_1\}\ def\ \{\psi_2\}$	{ top level function definition with pre- and post-conditions }
def	::=	$f\ (\overline{x_i :: \tau_i}) :: (\overline{y_j :: \tau_j}) = e$	{ function definition. Output results are named }
alt	::=	$C\ \overline{x_i :: \tau_i} \to e$	{ case branch }
τ	::=	α	{ type variable }
	|	$T\ \overline{\tau_i}$	{ type constructor application }

Figure 4: CAVI-ART IR abstract syntax

let bound variables in a nested sequence of **let** expressions are distinct, and also different to the function arguments.

In Fig. 5 we partially show the IR code for function *insertAVL*. In the **letfun** main expression, the code for functions *height*, *compose*, *leftBalance* and *rightBalance* is not shown. The first one computes the height of an AVL with a time cost in $O(1)$, by just getting the value stored in its root node. The second one just joins two trees and a value received as arguments to form an AVL tree. The other two are responsible for performing the LL, LR, RL and RR rotations that reestablish the height invariant of AVLs. In what follows, we will use the *insertAVL* function, together with all its auxiliary functions defined in the **letfun** expression, as a running example of UUT.

We define a *static path* through a set of mutually recursive functions defined together in an UUT, as a potential execution path starting at the top level function, and ending when this function produces a result. Not all static paths correspond to actual execution paths, since some static paths may be unfeasible. We define the *depth* of a static path, as the maximum number of unfoldings that a recursive function may undergone in the path. When all the recursive functions in the UUT are tail recursive, this definition of depth essentially corresponds to the number of iterations in imperative loops. Depth 1 corresponds to none iteration in all loops, depth 2 corresponds to at most one iteration, and so on. When there is at least one non-tail recursive function in the UUT, the depth of the path is the depth of the call tree deployed during the path execution, considering only the calls to the non-tail recursive function. Depth 1 means that each recursive function executes one of its base cases, depth 2 corresponds to that at least one recursive function has executed a recursive case, and then this recursive call has executed its base case, and so on.

In the *insertAVL* example there are two paths with depth 1. In the first one, the input tree is empty and the internal function *ins* immediately terminates by creating a new node with the value x and with height 1. In the second one, the value x being inserted is already present at the root of the input tree, and the *ins* function terminates without inserting anything. With depth 2, there are at least 4 paths through the function *ins*: two of them recursively call to *ins* with the left subtree as an argument, and another two call to *ins* with the right one as an argument. The rest of the path inside these recursive calls is one

```
define insertAVL (x::Int, t::AVL Int)::(res::AVL Int) =
  letfun
    ins (x::Int, t::AVL Int)::(res::AVL Int) =
      case t of
        leafA                                          -> nodeA x 1 leafA leafA
        nodeA y::Int h::Int l::(AVL Int) r::(AVL Int) ->
                        let b1::Bool = x < y in
                        case b1 of
                          true  -> let ia::(AVL Int) = ins x l in
                                   equil ia y r
                          false -> let b2::Bool = x > y in
                                   case b2 of
                                     true  -> let ib::(AVL Int) = ins x r in
                                              equil l y ib
                                     false -> t

    equil (l::AVL Int, x::Int, r::AVL Int)::(res::AVL Int) =
      let hl::Int  = height l in
      let hr::Int  = height r in
      let hr2::Int = hr + 2 in
      let b::Bool  = hl == hr2 in
        case b of
          true  -> leftBalance l x r
          false -> let hl2::Int = hl + 2 in
                   let b2::Bool = hr == hl2 in
                   case b2 of
                     true  -> rightBalance l x r
                     false -> compose l x r

    ...  definitions of height, compose, leftBalance and rightBalance
  in ins x t
```

Figure 5: CAVI-ART IR for function *insertAVL*

of the depth 1 paths. After inserting the value, the static paths go on with a call to function *equil*, and there are two paths in this function. Combined with the other 4, this gives 8 paths. Then, we should consider paths through *leftBalance* or *rightBalance*, depending on the branch taken by *equil*, and so on. The combinatorics soon produces an exponential number of paths when the UUT is as complex as in our example. Notice however, that many of these paths are unfeasible. For instance, when inserting a new node in the left subtree, it may not happen that *equil* takes the branch calling to *rightBalance*: if any unbalance arises after inserting to the left, it should be in the left subtree.

Given an UUT written in the IR language, and the user having fixed a maximum depth, our prototype generates all the static paths having a depth smaller than or equal to this maximum depth. For each static path, it collects all the equalities occurring in the **let** bindings traversed by the path, and the conditions that hold in each traversed **case** expression forcing the path to follow one of the branches. These equalities and conditions are converted into restrictions for the SMT solver. For instance, for the *insertAVL* depth-1 path in which the inserted element is already in the tree, the following constraints are collected:

$$(t = nodeA\ y\ h\ l\ r) \wedge (b1 = x < y) \wedge (b1 = false) \wedge (b2 = x > y) \wedge (b2 = false) \wedge (res = t)$$

Then, the solver is given the set of constraints corresponding to the UUT precondition, together with the set of constraints corresponding to a static path. If the solver finds a model, it means that it satisfies the precondition, and that the path is feasible. Then, the model assignment for the input arguments (in our example, for the tree t and for the value x) constitutes a test-case that, when run, it does not violate the precondition and exactly exercises that path.

```
(declare-const res (AVL  Int))
(declare-const ins_1_x  Int)
(declare-const ins_1_t (AVL  Int))
(assert (=  x  ins_1_x))
(assert (=  t  ins_1_t))
(declare-const ins_1_res (AVL  Int))
(assert (=  res  ins_1_res))
(declare-const ins_1_y  Int)
(declare-const ins_1_h  Int)
(declare-const ins_1_l (AVL  Int))
(declare-const ins_1_r (AVL  Int))
(assert (=  ins_1_t (nodeA  ins_1_y  ins_1_h  ins_1_l  ins_1_r)))
(declare-const ins_1_b1  Bool)
(assert (=  ins_1_b1 (<  ins_1_x  ins_1_y)))
(assert (=  ins_1_b1  true))
(declare-const ins_2_x  Int)
(declare-const ins_2_t (AVL  Int))
(assert (=  ins_1_x  ins_2_x))
(assert (=  ins_1_l  ins_2_t))
(declare-const ins_2_res (AVL  Int))
(declare-const ins_1_ia (AVL  Int))
(assert (=  ins_1_ia  ins_2_res))
(assert (=  ins_2_t leafA))
(assert (=  ins_2_res (nodeA  ins_2_x  1  ins_2_t  ins_2_t)))
(declare-const equil_1_l (AVL  Int))
(declare-const equil_1_x  Int)
(declare-const equil_1_r (AVL  Int))
(assert (=  ins_1_ia  equil_1_l))
(assert (=  ins_1_y  equil_1_x))
(assert (=  ins_1_r  equil_1_r))
...
```

Figure 6: Constraints for a path of depth 1 in *insertAVL*

In Fig. 6 we show the beginning of the set of constraints generated by our proptotype for a path of depth 1 that inserts an element to the left of the input tree. As there are two calls to function *ins*, its variables must be renamed in each call. For instance, `ins_2_t` stands for argument t of the second call to *ins*. For this path, the solver finds the model $t = nodeA\ 3\ 1\ leafA\ leafA$, $x = 2$ and $res = nodeA\ 3\ 2\ (nodeA\ 2\ 1\ leafA\ leafA)\ leafA$, which corresponds to a path inserting the value $x = 2$ to the left of the tree t, being its left subtree an empty tree.

We have prepared a UUT test suit for our prototype, including functions dealing with a variety of data structures, such as sorted arrays, sorted lists, binary search trees, AVL trees, leftist heaps, and red-black trees. The path depth was set to 3 for all of them except for *insertAVL*, which generated a two million lines constraint file, if depth were set to 3. The results are shown in Fig. 7. For complex UUTs such as this one, and also for the *quicksort* algorithm, and for the union of two leftist heaps, the number of unfeasible paths is very high. The insertion in a leftist heap is in fact a wrapper for the union function, and gives also a high number of unfeasible paths.

Each satisfiable path produces a test-case that exercises that path. If the SMT answered `sat` for all the UUT feasible paths, then we would have a test suit covering all the UUT paths up to the given depth. In our examples, we have obtained no `unknown` result. This means that all feasible paths have been converted into test-cases. For programs evaluating complex conditions in the **case** expressions, it may be the case that the solver cannot find a model, even if the set of constraints is satisfiable. In this situation, we would obtain neither `sat` nor `unsat` as the solver answer, but `unknown`.

UUT	depth	total paths	sat paths	unsat paths	unknown
binSearch	3	15	13	2	0
DutchNationalFlag	3	40	40	0	0
linearSearchArray	3	8	8	0	0
insertArray	3	8	8	0	0
quicksortArray	3	33	4	29	0
insertList	3	8	8	0	0
deleteList	3	12	12	0	0
searchBST	3	30	30	0	0
insertBST	3	30	30	0	0
searchAVL	3	30	30	0	0
insertAVL	2	66	10	56	0
searchLLRB	3	30	30	0	0
unionLeftist	3	354	8	346	0
insertLeftist	3	354	5	349	0

Figure 7: Paths generated for a suit of UUTs

5 Partial Automatic Verification

In prior sections, we have illustrated the use of SMTs for generating an exhaustive set of black-box test-cases up to a given size, and an exhaustive set of white-box ones covering all the UUT paths up to a given depth. Also, we have explained how to use SMTs to automatically check the validity of the results returned by the UUT. These results would allow us to perform exhaustive testing of programs by just pressing a button. The only price to be paid in order to get these benefits is writing formal preconditions and postconditions for the UUT. This may seem to be a big effort, since usually programmers are not very found to write formal specifications. In our case, these formal assertions are anyway mandatory because our CAVI-ART platform has been specifically designed to assist in the formal verification of programs. Someone may wonder why testing is needed in a verification platform, since verification gives in general more guarantees than testing. The answer to this question is that, when forced to write formal specifications, programmers usually write incomplete ones in their first attempts. So, automatic testing may help them to debug their initially weak specifications before embarking themselves into formal verification.

Using the same strategy as that of black-box test-case generation, we could improve a bit the exhaustiveness of white-box cases by generating *all* models for every satisfiable path. This would probably result in generating a huge number of cases. We have not followed this idea, but instead have followed an equivalent solution involving much less computational effort. We call it *partial automatic verification*.

The idea consists of adding to each satisfiable path a constraint expressing that the result returned in that path must satisfy the postcondition. Let us call $Q(\bar{x})$ to the precondition applied to the input arguments $\bar{x}$, $path(\bar{x},\bar{y})$ to the set of constraints collected by the path, which may depend on the input arguments and on some intermediate variables $\bar{y}$, and $R(\bar{z})$ to the postcondition applied to the result $\bar{z}$ returned by the path, where $\bar{z}$ is a subset of $\bar{x} \cup \bar{y}$. Then, we ask the solver to check whether the formula

$$(Q(\bar{x}) \wedge path(\bar{x},\bar{y})) \Rightarrow R(\bar{z})$$

is valid, i.e. all possible models satisfy it. Equivalently, we ask the solver to check whether the formula

$$Q(\bar{x}) \wedge path(\bar{x},\bar{y}) \wedge \neg R(\bar{z})$$

in unsatisfiable, because $\neg(A \rightarrow B) \equiv \neg(\neg A \vee B) \equiv A \wedge \neg B$. If it were so, then we would have proved that all the test-cases covering this path are correct. If we proved this kind of formulas for all the satisfiable paths up to a given depth, we would have proved the correctness of the UUT for all the possible executions exercising the UUT up to that depth. Since all this work is done without executing the UUT, in fact we are doing formal verification. And, as the automatic testing above described, by just pressing a button. In case we could automatically prove these formulas, we would forget about invariants, intermediate assertions, auxiliary lemmas, and all the effort usually implied by doing formal verification by hand.

If one of these formulas were satisfiable, then the models given by the SMT would constitute counter examples of correctness, i.e. test-cases for which the UUT will return an incorrect result. In this case, we would be doing debugging of our program at a very low cost.

We have applied this idea to the 10 satisfiable paths of *insertAVL* with $depth \leq 2$. The result has been that the formulas corresponding to the two depth-1 paths are `unsat`, the ones of two of the eight depth-2 paths are also `unsat`, and for the remaining formulas, the solver gives `unknown` as a result. That is, at least for 4 of these paths, the solver can prove their correctness. For the other 6, we conclude that the formulas are too complex for the current SMT technology. To return `unsat` is clearly more costly for the solver, since it must explore all the proof space. We remind that the recursive functions and predicates are internally transformed to universally quantified first order formulas, and that first order logic is undecidable in general. We did also the experiment of introducing a bug in the program by returning `nodeA x 0 leafA leafA` in the base case of *insertAVL*, instead of `nodeA x 1 leafA leafA`. Then the solver returned `sat` for the path formula, and returned the model $t = leafA$, which is in fact a failing test-case.

6 Related and Future Work

We have not found much literature for synthesizing test-cases from a precondition. The system Korat [4] was able to create relatively complex Java data structures such as binary search trees satisfying an executable JML assertion. The idea is similar to our previously cited work [2]: the system generates all possible trees and then filter out those satisfying the assertion.

A closer approach to the one presented here is the use of the Alloy system [8, 9] for syntesizing complex data structures. The system uses an specification language based on the language Z, and translates these specifications into a SAT formula. The language gives neither support for recursion nor for quantified formulas, but it includes sets, relations, and regular expressions. Integers are encoded in the SAT formula as bit sequences. With some specification effort, they have been able to synthesize red-black trees. Given the lack of support of SAT solvers for complex theories, the generated formulas need a huge number of boolean variables, so only small cases can be synthesized in a reasonable time.

Our protoype is still incomplete. We are able to generate constraints from pre- and postconditions, to parse our IR programs, and to generate constraints for paths. Also, to get exhaustive models for a given set of constraints. What we lack yet is the integration of all these pieces into a system interacting with the UUT and checking the results for validity. A picture of the final system is shown in Fig. 8.

The input to the tool is a CLIR file containing the UUT code written in the IR language, and its formal specification, written in the IR specification language. The CAVI-ART IR was thought as an intermediate

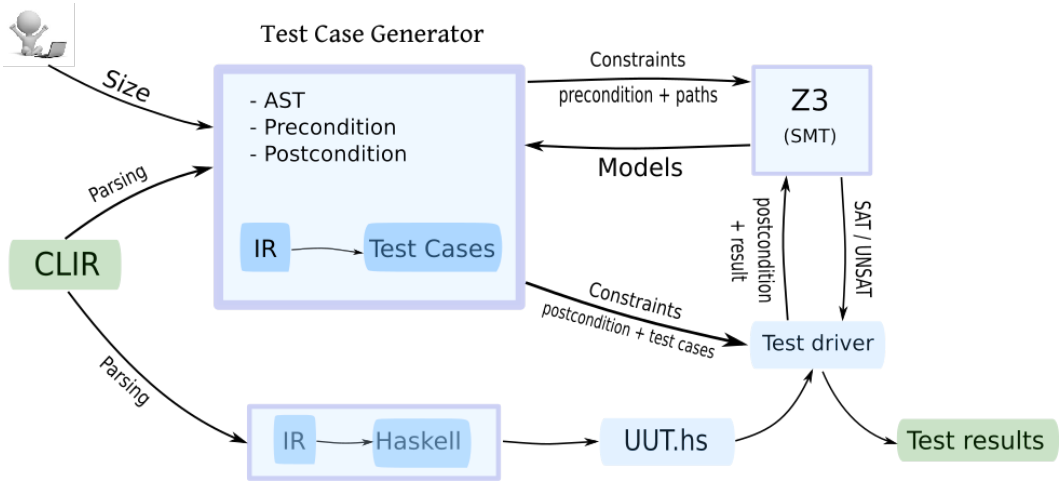

Figure 8: A picture of the CAVI-ART integrated testing system.

representation for the platform, and it is not directly executable. So, in the bottom part of the tool, there is a translation from the IR code to Haskell. The Test Driver is also Haskell code, and it is responsible for interacting with the UUT. It will deliver the tests-cases to the UUT, and it will receive and check the results the latter will produce. The left part of the tool is the TCG. It will receive from the user the desired size parameters and the desired depths for the static paths, it will convert the specifications and the paths into SMT constraints, and it will interact with the SMT solver. Then, it will translate the SMT results into Hashell test-cases. There will be an inverse translation in the Test Driver from the Haskell results returned by the UUT to the SMT syntax, in order to check whether the postcondition is satisfied, again with the help of the SMT solver.

We also plan to make the system generic in the sense that it could process definitions of new datatypes and predicates, and could generate test-cases for them. For the moment, we can deal with a fix set of datatype definitions for arrays, lists, and a number of different trees. These added features are also future work. When the system be complete, it will automatically generate test-cases, execute them, check the validity of the results, and will perform as many partial verifications as the solver would be able to deal with, giving as a result a fully automatic testing tool that will perform thorough testing of algorithms with complex preconditions at a very low cost.

References

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold & Phil McMinn (2013): *An orchestrated survey of methodologies for automated software test case generation*. Journal of Systems and Software 86(8), pp. 1978–2001, doi:10.1016/j.jss.2013.02.061. Available at `https://doi.org/10.1016/j.jss.2013.02.061`.

[2] Marta Aracil, Pedro García & Ricardo Peña (2017): *A Tool for Black-Box Testing in a Multilanguage Verification Platform*. In: *Proceedings of XVII Jornadas sobre Programación y Lenguajes, PROLE 2017, Tenerife, Spain, September 2017*, pp. 1–15.

[3] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia & Cesare Tinelli (2009): *Satisfiability Modulo Theories*. In Armin Biere, Marijn Heule, Hans van Maaren & Toby Walsh, editors: *Handbook of Satisfiability*,

Frontiers in Artificial Intelligence and Applications 185, IOS Press, pp. 825–885, doi:10.3233/978-1-58603-929-5-825. Available at `https://doi.org/10.3233/978-1-58603-929-5-825`.

[4] Chandrasekhar Boyapati, Sarfraz Khurshid & Darko Marinov (2002): *Korat: automated testing based on Java predicates*. In Phyllis G. Frankl, editor: *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24, 2002*, ACM, pp. 123–133, doi:10.1145/566172.566191. Available at `http://doi.acm.org/10.1145/566172.566191`.

[5] Koen Claessen & John Hughes (2000): *QuickCheck: a lightweight tool for random testing of Haskell programs*. In Martin Odersky & Philip Wadler, editors: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, ACM, pp. 268–279, doi:10.1145/351240.351266. Available at `http://doi.acm.org/10.1145/351240.351266`.

[6] Cormac Flanagan, Amr Sabry, Bruce F. Duba & Matthias Felleisen (1993): *The Essence of Compiling with Continuations*. In: *Proceedings of the conference on Programming Language Design and Implementation (PLDI'93)*, pp. 237–247, doi:10.1145/155090.155113. Available at `http://doi.acm.org/10.1145/155090.155113`.

[7] John Hughes (2009): *Software Testing with QuickCheck*. In Zoltán Horváth, Rinus Plasmeijer & Viktória Zsók, editors: *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures*, *Lecture Notes in Computer Science* 6299, Springer, pp. 183–223.

[8] Daniel Jackson, Ian Schechter & Ilya Shlyakhter (2000): *Alcoa: the alloy constraint analyzer*. In Carlo Ghezzi, Mehdi Jazayeri & Alexander L. Wolf, editors: *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000.*, ACM, pp. 730–733, doi:10.1145/337180.337616. Available at `https://doi.org/10.1145/337180.337616`.

[9] Sarfraz Khurshid & Darko Marinov (2004): *TestEra: Specification-Based Testing of Java Programs Using SAT*. *Autom. Softw. Eng.* 11(4), pp. 403–434, doi:10.1023/B:AUSE.0000038938.10589.b9. Available at `https://doi.org/10.1023/B:AUSE.0000038938.10589.b9`.

[10] Manuel Montenegro, Susana Nieva, Ricardo Peña & Clara Segura (2017): *Liquid Types for Array Invariant Synthesis*. In: *International Symposium on Automated Technology for Verification and Analysis, ATVA 2017*, Springer, pp. 289–306.

[11] Manuel Montenegro, Ricardo Peña & Jaime Sánchez-Hernández (2015): *A Generic Intermediate Representation for Verification Condition Generation*. In Moreno Falaschi, editor: *Logic-Based Program Synthesis and Transformation - 25th International Symposium, LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected Papers*, *Lecture Notes in Computer Science* 9527, Springer, pp. 227–243, doi:10.1007/978-3-319-27436-2. Available at `http://dx.doi.org/10.1007/978-3-319-27436-2`.

[12] Leonardo Mendonça de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In C. R. Ramakrishnan & Jakob Rehof, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, *Lecture Notes in Computer Science* 4963, Springer, pp. 337–340.

[13] Andrew Reynolds, Jasmin Christian Blanchette, Simon Cruanes & Cesare Tinelli (2016): *Model Finding for Recursive Functions in SMT*. In Nicola Olivetti & Ashish Tiwari, editors: *Automated Reasoning - 8th International Joint Conference, IJCAR 2016, Coimbra, Portugal, June 27 - July 2, 2016, Proceedings*, *Lecture Notes in Computer Science* 9706, Springer, pp. 133–151, doi:10.1007/978-3-319-40229-1_10. Available at `https://doi.org/10.1007/978-3-319-40229-1_10`.

[14] Takahisa Toda & Takehide Soh (2016): *Implementing Efficient All Solutions SAT Solvers*. *ACM Journal of Experimental Algorithmics* 21(1), pp. 1.12:1–1.12:44, doi:10.1145/2975585. Available at `https://doi.org/10.1145/2975585`.