

Automatically Identifying Sufficient Object Builders from Module APIs

Pablo Ponzio^{1,3}(✉), Valeria S. Bengolea¹, Mariano Politano^{1,3},
Nazareno Aguirre^{1,3}, and Marcelo F. Frias^{2,3}

¹ Universidad Nacional de Río Cuarto, Río Cuarto, Argentina
{pponzio,vbengolea,mpolitano,naguirre}@dc.exa.unrc.edu.ar

² Instituto Tecnológico de Buenos Aires (ITBA), Buenos Aires, Argentina
mfrias@itba.edu.ar

³ Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET),
Buenos Aires, Argentina

Abstract. Various approaches to software analysis (e.g. test input generation, software model checking) require engineers to (manually) identify a subset of a module’s methods in order to drive the analysis. Given a module to be analyzed, engineers typically select a subset of its methods to be considered as object builders to define a so-called *driver*, that will be used to automatically build objects for analysis, e.g., combining them non-deterministically, randomly, etc. This requires a careful inspection of the module and its API, since both the relative exhaustiveness of the analysis (leaving important methods out may systematically avoid generating different objects), as well as its efficiency (the different bounded combinations of methods grows exponentially as the number of methods increases), are affected by the selection.

We propose an approach for automatically selecting a set of builders from a module’s API, based on an evolutionary algorithm that favors sets of methods whose combinations lead to producing larger sets of objects. The algorithm also takes into account other characteristics of these sets of methods, trying to prioritize the selection of methods with less and simpler parameters. As the implementation of this evolutionary mechanism requires in principle handling and comparing large sets of objects, and this grows very quickly both in terms of space and running times, we employ an *abstraction* of sets of objects, called field extensions, that involves using the field values of the objects in the set instead of the actual objects, and enables us to effectively implement our mechanism. An experimental assessment on a benchmark of stateful classes shows that our approach can automatically identify sets of builders that are *sufficient* (can be used to create any instance of the module) and *minimal* (do not contain superfluous methods), in a reasonable time.

1 Introduction

As software is becoming more ubiquitous thanks to the rapid advances in technology, guaranteeing the functional correctness of software is more crucial than

ever. Thus, a research area of growing importance is that of automated software analysis, whose goal is to assist engineers, through the provision of tools for automated analysis, in finding deficiencies both in software and software related models. Automated test generation [1, 11, 13, 17, 24, 25, 28, 29, 32], software model checking [9, 34, 35], and static analyses [6, 16], among many others, are prominent approaches in this line of research.

While these techniques involve in many cases fully automated analyses, their application often requires some effort from the engineers. Software model checkers rely on the definition of *drivers*, programs that allow one to build inputs for the code under analysis. Similarly, in parameterized-unit testing approaches [33] a mechanism for building inputs is mandatory. Some symbolic execution based tools require the so-called “*object factories*” to build tests cases involving inputs with non-primitive types [32]. Automated test generation techniques based on a module’s API can be used for building inputs for non-primitive types [11, 24], thus automating the above-mentioned input-generation issues. But they usually present difficulties in generating a good set of diverse inputs for stateful, complex structures. This is even more difficult for structures with rich APIs [26]. Many authors have addressed this problem by defining different approaches for guiding test generation, to create more diverse sets of inputs [7, 26].

In this paper, we take a different approach to address the problem of generating better inputs for stateful modules. We observe that the selection of routines from a module API, to feed an input generation tool so as to build input structures for program analysis (drivers for model checking, input structures for parameterized unit tests, etc.), has a crucial impact on the analysis. We call *builders* a set of routines B , drawn from a module’s M API, that can be employed to create input structures in an automated program analysis for M (e.g. a driver for model checking). Clearly, the higher the number of different structures that can be created with B , the better the chances to find bugs in M . As the number of instances of a software module is potentially infinite, and the program analyses we target are also limited in the number of structures they can employ, we limit ourselves to a bounded-exhaustive set of structures for M [4] (e.g. all the instances of a linked list with up to k nodes). We denote this set by $BE(M, k)$. We say that a builders are *sufficient* if they can combined to build all the instances in $BE(M, k)$. Thus, sufficient builders are the best possible choice for bug finding (in a bounded setting). Notice that B can contain superfluous routines. A superfluous routine s is such that $BE(M, k)$ can be built using routines in $B - \{s\}$ (the simplest example being routines that never change the state of their parameters). These routines provide no benefits in terms of bug finding capabilities of the analysis. We call *minimal* a set of builders with no superfluous routines. Minimality is important because providing an analysis tool with superfluous routines often negatively impacts its efficiency (the number of ways k routines can be combined usually increases exponentially with k).

Manually selecting sufficient and minimal builders is not an easy task: it requires a thorough analysis of the available routines and a deep understanding of the program semantics. This is especially hard for programs with rich APIs,

where there are many routines and a lot of redundancy in the API (see Sect. 2). We propose an automated approach for identifying such a sufficient and minimal set of builders, based on an evolutionary algorithm that searches for a minimal set of routines that is capable of generating the maximum number of different (bounded) objects (i.e., $BE(M, k)$). Moreover, our evolutionary approach also takes into account other characteristics of the builders, such as the number and complexity of their parameters, so that “simpler” routines are favored in the search. The goal is to choose builders that can be more easily and more efficiently used by the subsequent program analyses.

The fitness value for a set of routines R is based on the number of bounded structures that can be generated using combinations of these routines. To compute the fitness we use a modified version of a random test case generation tool (Randoop [24]) to generate as many bounded structures as possible from R , allowing at most k of objects of each type in the structures (a parameter to our algorithm). As sets of objects are very expensive to maintain and manipulate, both in terms of space and running time, we employ an efficient abstraction of a set of objects, called *field extensions*, defined as the set of field values appearing in any of the objects in the set [25]. Thus, instead of counting the number of different objects achieved by a candidate, the fitness function will compute the field extensions as objects are generated, and return the number of field values in the extensions. Intuitively, a higher number of field values in the field extensions means that the builders can be used to construct a more diverse set of objects, and therefore they should be preferred over other sets of builders.

We assess our approach experimentally on a benchmark of stateful Java classes drawn from the literature. The results show that in our case studies our approach identifies sets of routines that are sufficient and minimal, in a reasonable time. We also assess the impact of our approach in an automated analysis, namely, in the generation of test cases for parameterized tests. We compare how the random test case generation tool Randoop behaves when fed with the full module API, against providing the tool with only the builders identified by our approach. The results indicate that in the latter case Randoop generated more (and larger) objects, within a fixed time budget.

2 Motivating Example

In this section, we motivate our approach by means of a running example. The Apache NodeCachingLinkedList (NCL for short) [36] consists of a main circular doubly linked list, that holds the elements of the collection, and a secondary singly linked list that acts as a cache for nodes that have been removed from the main list. Nodes stored in the cache can be reused, and added again to the main list when inserting elements in the main list. Thanks to its cache, in applications where insertions and removals from the list are very frequent, NCL can significantly reduce the overhead needed for memory allocation and garbage collection of nodes. As an illustration, Fig. 1 shows the three NCL instances that can be built with exactly two nodes.

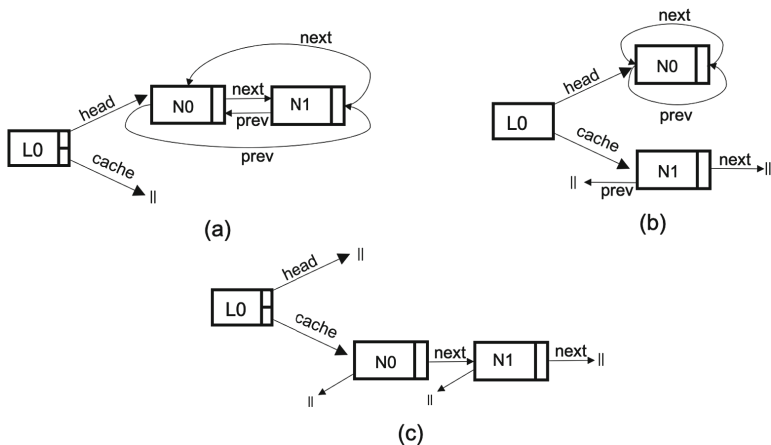


Fig. 1. Three NodeCachingLinkedList instances with exactly two nodes

Table 1. Apache's NodeCachingLinkedList API

No.	Return type	Method name	Obs?	No.	Return type	Method name	Obs?
0		NCL()	no	17	boolean	isEmpty()	yes
1		NCL(int)	no	18	Iterator	iterator()	no
2		NCL(Collection)	no	19	int	lastIndexOf(Object)	yes
3	boolean	add(Object)	no	20	ListIterator	listIterator()	no
4	void	add(int,Object)	no	21	ListIterator	listIterator(int)	no
5	boolean	addAll(Collection)	no	22	Object	remove(int)	no
6	boolean	addAll(int,Collection)	no	23	boolean	remove(Object)	no
7	boolean	addFirst(Object)	no	24	boolean	removeAll(Collection)	no
8	boolean	addLast(Object)	no	25	Object	removeFirst()	no
9	void	clear()	no	26	Object	removeLast()	no
10	boolean	contains(Object)	yes	27	boolean	retainAll(Collection)	no
11	boolean	containsAll(Collection)	yes	28	Object	set(int,Object)	no
12	boolean	equals(Object)	yes	29	int	size()	yes
13	Object	get(int)	yes	30	List	subList(int,int)	no
14	Object	getFirst()	yes	31	Object[]	toArray()	yes
15	Object	getLast()	yes	32	Object[]	toArray(Object[])	yes
16	int	indexOf(Object)	yes	33	String	toString()	yes

NCL has a very rich API, as shown in Table 1. However, for building any feasible NCL object only a few methods from the API suffice. For example, combinations of the methods in Fig. 1.1, when instantiated with appropriate parameters, can be used to build any desired (finite) NCL object. Thus, the methods therein are an example of a sufficient set of builders. Notice that, after using the constructor, the main list of NCL can be populated just by using the `addFirst` method. However, if we want to generate instances where the cache is not empty, we can do so through the `removeFirst` method, as the sufficient set of builders suggests. For most automated analyses, we would like to consider as varying scenarios (inputs) as possible, hence the motivation to build sufficient sets of builders. Furthermore, the builders in Fig. 1.1 are also minimal, since the lack of any one of them would imply that some NCL's objects cannot be constructed anymore with the routines.

```
(0) NodeCachingLinkedList ()
(7) addFirst (Object)
(25) removeFirst ()
```

Figure 1.1. A sufficient set of builders for NCL

```
(3) add (Object)
(4) add (int , Object)
(7) addFirst (Object)
(8) addLast (Object)
```

Figure 1.2. Add variants that can be used to populate NCL’s main list

Notice that there can be many sets of sufficient and minimal builders. For example, we get sufficient and minimal builders by replacing `addFirst` in Fig. 1.1 with any of the other add variants shown in Fig. 1.2, as for any way of filling up NCL’s main list with `addFirst` there exists a different way to build the same object using another add variant (perhaps invoked with different parameters and changing the execution order).

We also observe that the simpler the parameters of a routine, the easier to use the routine is for generating inputs in the context of a program analysis. For instance, among the alternative add routines for NCL (Fig. 1.2), `add(int, Object)` receives more parameters than the other three methods, therefore it is harder to generate parameters for it when generating inputs. This makes the other three alternatives preferred over it. Thus, our approach takes into account the number of parameters and their complexities for selecting the best possible builders.

Many methods in Table 1 are marked as observers (column Obs?), meaning that they do not modify the objects they operate on, nor they are useful for creating non-primitive objects. Hence, observers are always superfluous, and should never be included in a set of minimal builders. Our approach tries to recognize them beforehand, and discards them from the search to significantly reduce the search space.

To conclude this section we remark that, when fed with the whole NCL’s API, our approach automatically identified the sufficient and minimal set of builders for NCL shown in Fig. 1.1.

3 Background

3.1 Field Extensions

The idea behind field extensions [25] is to define a representation for a set of objects that is smaller in size and easier to manipulate algorithmically. This representation implies some loss of information, but for certain applications (like the one in this paper) they are precise enough to be useful in practice [1, 12, 25, 26, 29].

```

head = (L0, null), (L0, N0)
cache = (L0, null), (L0, N1), (L0, N0)
next = (N0, N1), (N1, N0), (N0, N0), (N1, null)
prev = (N0, N1), (N0, N0), (N1, null), (N0, null)

```

Figure 1.3. Field extensions for the set of instances in Fig. 1

Given a set S of objects, its field extensions representation consist of a set of pairs for each field f , such that (obj, val) belongs to the field extensions of f if $\text{obj}.f = \text{val}$ (i.e., the value of f for obj equals to val), for some object obj in S . As an example, consider the instances displayed in Fig. 1. Its corresponding field extensions are shown in Fig. 1.3. We omit the values stored in the nodes for the sake of clarity. Notice that structure (a) in Fig. 1 can be built using only add methods, whereas for (b) and (c) we have to also employ some kind of remove operation, to move nodes from the main list to the cache. Notice that values (L_0, N_0) and (L_0, N_1) for the cache field only appear in the field extensions when the structures have nodes in the cache, like (b) and (c). In addition, prev fields of nodes in the cache are always null , but prev fields can never be null in the main list (due to its circularity). This means that field extensions for structures that have non-empty caches have the potential of having a larger number of values than those for structures with no caches.

It is important to canonicalize structures before computing field extensions [12]. Canonicalization involves assigning unique identifiers N_0, N_1, \dots to each of its nodes during a traversal of the structure (we employ a breadth first traversal), starting at the root. Nodes visited first receive smaller identifiers than those visited afterwards during the traversal. Fields must be visited in a fixed order. Note that structures in Fig. 1 are all in canonical breadth-first form.

3.2 Random Test Case Generation

Random test generation consists of randomly producing inputs in order to test software [8, 21, 24]. Random input generation is straightforward when considering basic (numeric) data types, but producing inputs of other more complex types, in particular instances of *stateful* classes, is less obvious and calls for a more complex mechanism, other than just using random number generators. One such mechanism, that has been implemented by various tools for random test generation for object-oriented code, is based on randomly combining method sequences, that produce inputs of different types [8, 21, 24]. The process associated with the Randoop tool [24] that we use here, works essentially as follows. For every datatype, a set of sequences that produce inputs of such datatype, is maintained. To start with, for basic data types, a set of initial values is considered, and for class types, only null is considered at first (these can be considered test sequences of size one). The procedure to build a new test sequence starts by randomly selecting a method m , among all methods in the software under test. For example, one could randomly choose one of the methods for the NCL's API (Table 1), say $\text{add}(\text{Object})$. To actually build the test sequence, values for

each of the parameters of the method m , of the corresponding types, have to be provided. These are obtained by randomly selecting test sequences, from the sets of sequences of the corresponding types, and sequentially composing them, with method m as a last statement. As an example, say that a sequence containing only the constructor of NCL is randomly selected, from the available sequences for the NCL type, and for the parameter of `add`, an Integer with value 0 is randomly chosen. Combining all these sequences together results in:

```
NodeCachingLinkedList l = NodeCachingLinkedList ();  
l.add(new Integer (0));
```

This new sequence can now be stored for later use as a parameter for other methods that operate on NCL objects.

This process is repeated until either a time budget is exhausted, or the desired number of tests (set by the user) is generated. Randoop uses guidance from the execution of tests to avoid generating illegal tests. We refer the interested reader to the article introducing Randoop [24], for further details.

An important issue to remark here is that the execution of each test sequence generated by Randoop produces a number of objects for the given type (NCL in the example). We exploit this characteristic of Randoop to compute the fitness function for a set of methods, although instead of storing actual objects we will maintain field extensions, as we explain in more detail in Sect. 4.

4 An Evolutionary Algorithm for Identifying Sufficient Object Builders

As mentioned before, to find a sufficient set of builders from a program API we design a genetic algorithm, that we describe below. Genetic algorithms [14] are non-exhaustive guided search algorithms, based on a hill climbing strategy [30]. The search space is composed of a generally very large set of individuals (the candidates), and the search objective is to find an individual with sought-for features. As opposed to classic search algorithms, genetic algorithms maintain a set of individuals, called the population, and search progresses by iteratively selecting a number of individuals in the population, using these for evolution (building new individuals out of these), and leaving out some individuals of the whole set (the “old” ones and the “new” ones). Selection of individuals for population evolution, as well as individuals’ removal, are guided by a fitness function, the heuristic function used to guide the search. This function applies to individuals, and its result is generalizable to the population too (e.g., the fitness of the population may be taken as the fitness of its “fittest” individual). This function captures the features sought for in the search, and thus can be used as a halting criterion (e.g., algorithm stops after finding an individual with fitness above a certain threshold). Finally, individuals are often called chromosomes, and represented as vectors of genes that capture their characteristics. This idea is strongly related to how new individuals are constructed: by representing candidates as

vectors of independent characteristics, one can build new candidates by combining part of the characteristics of an individual with part of the characteristics of another, or by arbitrarily changing a characteristic of a given individual. These two forms of evolution are called crossover and mutation, respectively, and are the traditional mechanism to build new candidates out of existing ones in genetic algorithms. For further details, we refer the reader to [22].

4.1 Chromosome Representation

In the context of our problem, candidate solutions represent sets of methods from the API of the module being analyzed. We then employ vectors of boolean values as chromosome representation. Let n be the number of methods in the API; the chromosomes in our algorithm will be vectors of size n . For any vector, the i -th position is true if and only if the chromosome contains the i -th method of the API. For example, there are 34 methods in the NCL’s API (Table 1), and we enumerated them from 0 to 33. The sufficient set of builders in Fig. 1.1 is characterized by the vector with positions 0, 7 and 25 set to true, and the remaining positions set to false. In this case, the whole search space consists of the 2^{34} possible chromosomes.

4.2 Fitness Function

Given a chromosome representing a set of methods M , our fitness function computes an approximation of the number of bounded objects that can be built using combinations of methods in M . Chromosomes with higher fitness values are estimated to build more objects than those that have smaller fitness values.

Ideally, we would like to explore all the feasible objects within a small bound k , that can be built using the methods of the current chromosome, i.e., $BE(M, k)$. In other words, we need a bounded exhaustive generator for the set of methods. The bound k represents the maximum number of objects that can be created for each class (in Fig. 1, the number of nodes in the NCL objects are bounded by $k = 2$), and the maximum number of primitive values available (for example, integers from 0 to $k - 1$). For this purpose, we developed a prototype modifying the Randoop tool, discussed briefly in Sect. 3.2. First, we altered Randoop to work with a fixed set of primitive values (integers from 0 to $k - 1$). (Normally, Randoop would save primitive values that are returned by the execution of tests, and reuse these values in future tests.) Second, we make Randoop drop sequences of methods that create objects with more than k objects (of any type), to stop it from building objects larger than needed. To achieve this, we canonicalize the objects generated by the execution of each sequence, and we discard the sequence if some object has an index equal or larger than k . Third, we extend Randoop with “global” field extensions, and when the execution of a sequence terminates all the field values of the objects generated by the sequence are added to the field extensions. For example, if Randoop had generated the objects in Fig. 1, then the global field extensions would have the values shown in Fig. 1.3. Our goal is that, given a bound k , when our modified version of

```
(0) NodeCachingLinkedList ()
(7) addFirst (Object)
(8) addLast (Object)
(25) removeFirst ()
```

Figure 1.4. A set of sufficient but not minimal builders for NCL

```
(0) NodeCachingLinkedList ()
(4) add (int , Object)
(23) remove (Object)
```

Figure 1.5. Sufficient and minimal builders for NCL with more complex parameters than the ones in Fig. 1.1

Randooop terminates the global field extensions contain all the field values of the bounded exhaustive set of structures with up to k nodes, $BE(M, k)$. The result of the fitness function for the chromosome is the number of field values in the global extensions computed by the tool.

Our rationale for using bounded sets of objects is akin to the small scope hypothesis for bug finding [2]: if one set of methods cannot be used to build small objects that allow to differentiate it from another set of methods, then it is unlikely that these two sets can be distinguished with larger objects. This hypothesis held during our empirical evaluation across all our case studies.

We found that, besides being affected by chance, our tool rarely misses building objects that should add relevant values to the global extensions, when small values for k are employed.

Choosing Better Sets of Builders. In this section, we propose two ways to improve our evolutionary algorithm by tailoring the fitness function to obtain better sets of builders. This is strongly motivated by the way builders are used to build inputs in program analysis. On the one hand, if we have two sufficient set of builders, the set with the smaller number of methods should always be preferred. In this context, there is no reason to include superfluous methods in builders. For example, the builders in Fig. 1.4 can be used to create the same NCL objects as the builders in Fig. 1.1 of Sect. 2 (both sets are sufficient), but they are not minimal since `addLast` is superfluous.

On the other hand, builders with more parameters, or more complex ones, are more taxing on program analysis, as they require more effort to be adequately instantiated. Thus, we define a simple criterion of parameter complexity and adapt our fitness to favor builders with simpler parameters over the more complex ones. For example, both sets of builders in Figs. 1.1 and 1.5 are sufficient and minimal (with 3 routines each), but builders in Fig. 1.5 have more parameters that need to be instantiated. Comparing Figs. 1.1 and 1.5 we can observe that `addFirst` has been replaced by `add`, which has an additional integer parameter, and that `removeFirst` was interchanged with `remove`, which possesses a

non-primitive parameter of type Object. Following the criteria explained above, we would like our algorithm to choose the set in Fig. 1.1 over that of Fig. 1.5.

Incorporating these ideas, the fitness function of our approach is defined by:

$$f(M) = \#fieldExt(M) + \left(\frac{w_1 * \left(1 - \frac{\#M}{\#MT}\right) + w_2 * \left(1 - \frac{(\#PP(M) + w_3 * RP(M))}{(\#PP(MT) + w_3 * RP(MT))}\right)}{w_1 + w_2} \right)$$

For a chromosome representing a set M of methods, drawn from the whole set of available methods of the API, MT , the most important part of the fitness for M , is the number of values in the field extensions, $\#fieldExt(M)$, that can be generated using our custom Randoop tool as explained in the previous section. The summand on the right implements the ideas presented in this section. It returns a real value in the interval $[0, 1]$ that is useful to break ties for sets of methods that generate field extensions with the same number of values. In the dividend, the first summand penalizes sets with larger numbers of methods, by computing the quotient of the number of methods in M to the number of methods in MT , and subtracting the result to 1. Constant w_1 ($w_1 \geq 1$) allows us to increase/decrease the weight of this summand with respect to the other summand. The second summand in the dividend penalizes sets of methods with more complex parameters. Similarly to w_1 , constant w_2 ($w_2 \geq 1$) serves the purpose of increasing/decreasing the weight of this factor in the sum. Notice that we sum up the parameters differently depending on their types: each primitive parameter adds 1 ($PP(M)$ is the number of primitive parameters in the methods of M), and each reference parameter adds a constant w_3 ($w_3 \geq 1$, $RP(M)$ is the number of reference-typed parameters in the methods of M), which allows us to increase the weight of reference parameters with respect to primitive ones. Intuitively, the whole right-hand summand computes the ratio between the number of parameters of M (with added weight for reference parameters) to the number of (weighted) parameters for MT . The result is then subtracted from 1. Finally, we divide by $w_1 + w_2$ to obtain the desired number in the interval $[0, 1]$.

In our experimental assessment we set $w_1 = 2$, $w_2 = 1$, $w_3 = 2$. These values were good enough for our approach to produce sufficient and minimal sets of builders in all our case studies.

It is important to remark that the presented criteria for choosing better builders is based on the kind of program analyses we target (generation of tests cases for parameterized tests, software model checking). New criteria can be defined with other goals in mind, and our approach can be adapted to support them by modifying the fitness function as we did in this section.

4.3 Overall Structure of the Genetic Algorithm

The previously described elements are the constituting parts of the genetic algorithm implementing our approach. A pseudocode of the genetic algorithm is shown in Algorithm 1. Notice that Algorithm 1 follows the general structure of

Algorithm 1. Genetic Algorithm implementing our approach

```
1:  $pop \leftarrow$  chromosomes with exactly one true gene
2: for  $i = 1 \dots numEvo$  do
3:    $pop \leftarrow$  keep the  $popSize$  fittest chromosomes from  $pop$ 
4:   for  $j = 1 \dots cRate * popSize$  do
5:      $c1, c2 \leftarrow$  select two random chromosomes from  $pop$ 
6:      $new \leftarrow$  single point crossover  $c1, c2$ 
7:     add  $new$  to  $pop$ 
8:   end for
9:   for  $c \in pop$  do
10:     $new \leftarrow$  mutate each gene of  $c$  with probability  $mRate$ 
11:    if  $new \neq c$  then
12:      add  $new$  to  $pop$ 
13:    end if
14:   end for
15: end for
16:  $result \leftarrow$  fittest chromosome of  $pop$ 
```

a genetic algorithm. The initial population is generated by producing all the feasible chromosomes with only one available method (vectors with false in all positions except one, set to true) (line 3). Then, it starts to iteratively evolve the population (lines 4–15). At the beginning of each evolution iteration, the algorithm discards some individuals to control population size, by keeping the $popSize$ fittest individuals of the current population and discarding the rest (line 5). Then, the algorithm performs single-point crossover on randomly selected individuals (lines 6–10). Crossover is applied a number of times that is proportional to the population size $popSize$, determined by the product of $popSize$ and the crossover rate parameter $cRate$ ($0 \leq cRate \leq 1$). Then, the algorithm mutates individuals (lines 11–15) by changing the value of each of its genes with probability $mRate$ ($0 \leq mRate \leq 1$). Any newly created individual by the crossover and mutation operations are added to the population.

The algorithm stops after $numEvo$ evolutions, with $numEvo$ a parameter of the algorithm. Notice that, we don't have a target value for our fitness, since an untried set of methods might produce a larger number of field extensions than the algorithm has currently seen. Again, there is a compromise to be made for choosing a good value for $numEvo$: a larger number increases the precision of the algorithm but increases its running time, whereas a smaller number makes it run faster but it might not result in the best set of builders.

As usual, we found a number for the parameters of our algorithm that seems to work well in practice. In our experimental evaluation, we set $numEvo = 20$, $popSize = 30$, $cRate = 0.35$, $mRate = 0.08$ (the last two are the default for the JGap library).

Most of Algorithm 1 is a default evolutionary implementation of the JGap Java library [37]. Notice that, if we take away the complexity of the fitness function, our evolutionary algorithm is rather standard, so it is not surprising that

an existing implementation works well for our purposes. Of course, improvements to the evolutionary algorithm, and fine tuning for its parameters (e.g., crossover/mutation rate) might yield faster execution times.

We also implemented a simple multi-threaded version of our approach, that helps improving its performance. Basically, at each iteration we make t copies of the current population, where t is the number of available threads, and evolve each of the population replicas independently of the others. After all the threads have finished, we keep the $100/t$ fittest individuals of the population evolved by each thread, and use them to build the population for the next iteration of the algorithm.

4.4 Reducing the Search Space by Observers Classification

We say a routine is an observer if it never modifies the parameters it takes, and never generates a non-primitive value as a result of its execution. Column `Obs?` in Table 1 (Sect. 2) indicates whether each NCL method is an observer or not. Clearly, an observer cannot be used to modify nor build new objects, and therefore can never belong to a minimal set of builders. Hence, if we can classify them correctly beforehand, we can remove the observers from the search to significantly reduce the search space, without losing precision. For example, in the NCL API (Table 1) there are 13 observers out of 34 methods, so by removing observers we prune more than one third of the search space.

To detect observers we run another customized Randoop version before our evolutionary algorithm. This time, we check for each method whether it modifies its inputs at each test sequence generated by Randoop involving the method, by canonicalizing the objects before and after execution of the method, and checking if the field values of the objects change after execution. If this is the case, the method is marked as a builder (not an observer). For return values, if in any test sequence generated by Randoop the method returns a non-primitive value, then we mark it as a builder as well. We run this custom Randoop until it generates a large number of scenarios for each method. Ten to twenty seconds was enough for our case studies. At the end of the Randoop execution, methods not marked as builders are considered observers and discarded before invoking the evolutionary algorithm.

Other approaches exist for the detection of pure methods [15, 31] (similar to our observers). Note that our evolutionary algorithm is not dependent on the method classification algorithm, so any of them could be useful for our purposes.

5 Experimental Results

In this section, we experimentally assess our approach. The evaluation is based on a benchmark of data structure implementations, including: NCL from Apache Collections [36]; `BinaryTree`, `BinomialHeap`, `FibonacciHeap`, `RedBlackTree` taken from [35]; `UnionFind`, an implementation of disjoint sets taken from JGraphT [38]. We also evaluate our technique on components of real software projects

such as `Lits` from the implementation of `Sat4j` [3], taken from [20], which consists of a variable store that monitors when a guess was last made about a value of a variable, and whether listeners are watching the state of that variable; and `Scheduler`, an implementation of a process scheduler taken from [10]. All the experiments were run on 3.4 GHz quad-core Intel Core i7-6700 machines with 8 GB of RAM, running GNU/Linux.

The evaluation consists of two parts. First, we ran our approach (Algorithm 1) on the whole module APIs of the aforementioned classes, to compute sets of builders for each case study. The goal is to assess how good are the builders identified, and the time it takes our approach to compute them. For each case study we ran our approach 5 times. The results are shown in Table 2, including the number of routines in the whole API (`#API`), a sample of identified builders (some methods might be interchanged in different runs, e.g., `addFirst` and `addLast` in `NCL`), and the average running time (in seconds) of the 5 runs. We manually inspected the results, and found that the automatically identified sets of builders were in all cases sufficient (all the feasible objects for the structure can be constructed using the builders) and minimal (do not contain superfluous methods). The approach is reasonably efficient, taking about 30 min in the worst case.

The second part of the evaluation regards how helpful are the identified builders in the context of a program analysis, namely, the automated generation of test cases. These objects might be used, for example, as inputs in parameterized unit tests. For the case studies that provide mechanisms to measure the size of objects and to compare objects by equality (i.e., the `size` and `equals` methods of data structures), we generated tests with Randoop using all the methods available in the API (`API`), and then we generated tests with Randoop using only the builder methods (`BLD`) identified by our approach in the previous experiment (Table 2). We then compare the number of different objects (`No. of Objs.`), and the size of the largest object (`Max Obj. Size`) created by the tests generated from the `API`, against the tests generated using methods from `BLD` only. We set three different test generation budgets: 60, 120 and 180 seconds (`Budget`). The results are summarized in Table 3. In addition, we consider another approach, `API+`, that involves the generation of tests using the `API` for a budget that encompasses the test generation budget (`Budget`) plus the time it takes our approach to identify builders for the corresponding case study. The results show that in the same test budget `BLD` generates in average 1280% more objects than `API`. Furthermore, when builders identification time is added to the test generation budget for `API` (`API+`), `BLD` can generate 568% more objects in average (w.r.t `API+`). In all cases, `BLD` also generates significantly larger objects than `API` and `API+`. In view of these results, it is clear that automated builders identification pays off for the automated generation of structures for stateful classes.

The experiments can be reproduced by following the instructions in the paper website [27]. Furthermore, in the site we experimentally show that the builders identified by our approach can be employed to build efficient drivers for software model checking. We don't show these results here due to space constraints.

Table 2. Builders computation results

	Sample Builders	Time
NCL	NCLinkedList(int)	1744
#API: 34	addFirst(Object)	
	removeFirst()	
UFind	UnionFind()	215
#API: 9	addElement(int)	
	union(int,int)	
FHeap	FibonacciHeap()	72
#API: 7	insert(int)	
	removeMin()	
RBT	TreeMap()	73
#API: 8	put(int)	
BTree	BinTree()	73
#API: 7	add(int)	
BHeap	BinomialHeap()	121
#API: 10	insert(int)	
Lits	Lits()	1229
#API: 26	getFromPool(int)	
	forgets(int)	
	setLevel(int,int)	
	setReason(int)	
Sched.	Schedule()	377
#API: 10	addProcess(int)	
	blockProcess()	
	quantumExpire()	

Table 3. Assessment of using the identified builders (BLD) vs the whole API (API) in test case generation. API+ involves test case generation with the whole API, with budget = (Budget + builders computation time)

	Budget	Max Obj. Size			No. of Obj.		
		API	BLD	API+	API	BLD	API+
NCL	60	8	16	11	1442	42021	13119
#API: 34	120	8	18	11	2423	69017	13247
#BLD: 3	180	9	18	11	3166	91647	13505
UFind	60	8	13	9	3388	34250	8351
#API: 9	120	9	13	9	5180	56418	8574
#BLD: 3	180	9	13	9	6695	74425	9387
FHeap	60	11	15	12	6989	32639	11499
#API: 7	120	12	17	13	11447	54264	17202
#BLD: 3	180	12	17	13	15344	72413	20775
RBT	60	8	15	8	1812	23034	3041
#API: 8	120	8	15	8	2678	35635	3698
#BLD: 2	180	8	15	8	3358	44807	3940
BTree	60	8	15	8	3600	24908	6019
#API: 7	120	8	15	8	5471	39239	7387
#BLD: 2	180	8	15	9	6975	50671	9247
BHeap	60	9	26	10	3874	65915	8076
#API: 10	120	10	29	10	5970	111402	9708
#BLD: 2	180	10	29	11	7638	147260	10606

6 Related Work

As mentioned throughout the paper, the problem of identifying sufficient builders is recurrent in various program analyses, including but not limited to software model checking and test generation. In works like [18,23], in the context of software model checking, and [5,24,32,33], in the context of automated test generation, and just to cite a few, the problem of identifying part of an API and provide it for analysis is present. Typically the problem is dealt with manually.

The use of search-based techniques to solve challenging software engineering problems is an increasingly popular strategy, which has been applied successfully to a number of problems, including test input generation [11], program repair [19], and many others. As far as we are aware of, this is a novel application of evolutionary computation in software engineering. An approach that tackles a related, but different, problem, is that associated with the SUSHI tool [5]. The aim with SUSHI is to feed a genetic algorithm with a *path condition*, produced by a symbolic execution engine, so that an input satisfying the provided path condition can be reproduced using a module’s API. This approach assumes that the API (or the subset of relevant methods) is provided, as opposed to our work, that precisely tackles the provision of the restricted API.

Our technique requires a mechanism for identifying *observers*, which we have solved within the work in the paper, resorting to random test generation,

and instrumentation for state monitoring. Approaches to the identification of observers, or more precisely *pure methods*, exist in the literature [15, 31]. Regarding these lines of work, notice that the focus of our evolutionary algorithm is not the identification of observers, but the construction of minimal and sufficient set of builders. Moreover, our approach is in fact independent of the mechanism used to identify observers/pure methods, and thus could be combined with the works just cited (i.e., replacing our random testing based approach by an alternative one).

7 Conclusions

In this work, we presented an evolutionary algorithm for automatically detecting sets of builders from a module’s API. We assessed our algorithm over several case studies from the literature, and found that it is capable of precisely identifying sets of builders that are sufficient and minimal, within reasonable running times. To the best of our knowledge, this is the first work that addresses this problem, which is typically dealt with manually.

We also showed preliminary results indicating that our approach can be exploited by test case generation tools to yield larger and more diverse objects. Other techniques, like software model checking, can benefit as well by using the identified set of builders to automatically construct efficient drivers. More experimentation needs to be done, but given the results in this paper our approach looks very promising.

One of the biggest challenges of this work was the construction of a tool to allow us to generate all the bounded structures, for a given maximum number k of objects, from the methods of the program API. The proposed solution worked well enough for our case studies, but avoiding randomness in the process would be desirable. Using bounded exhaustive generation tools rather than random generation would better fit our purposes [4], but unfortunately none of the tools for bounded exhaustive test generation produce inputs from a module’s API. We believe that a promising research direction, that we plan to further explore in future work, is to adapt our presented approach for bounded exhaustive test generation.

Some aspects of our genetic algorithm can be further improved. For instance, a more powerful classification for argument types, in the prioritization of methods according to their complexities, can be defined. Moreover, one may also incorporate other dimensions, such as *code complexity*, to favor simpler methods. We will explore this direction as future work. Also, our genetic algorithm implementation is, for most parts, a default evolutionary implementation of the JGAP Java library [37]. Of course, improvements to the evolutionary algorithm, and fine tuning for its parameters (e.g., crossover/mutation rate) might yield faster execution times, so we plan to investigate this further in future work.

References

1. Abad, P., et al.: Improving test generation under rich contracts by tight bounds and incremental SAT solving. In: Sixth IEEE International Conference on Software Testing, Verification and Validation, ICST 2013, Luxembourg, Luxembourg, 18–22 March 2013, pp. 21–30 (2013)
2. Andoni, A., Daniliuc, D., Khurshid, S.: Evaluating the small scope hypothesis. Technical report, MIT Laboratory for Computer Science (2003)
3. Berre, D.L., Parrain, A.: The Sat4j library, release 2.2, system description. *J. Satisf. Boolean Model. Comput.* **7**, 59–64 (2010)
4. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on Java predicates. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2002, pp. 123–133. ACM, New York (2002)
5. Braione, P., Denaro, G., Mattavelli, A., Pezzè, M.: SUSHI: a test generator for programs with complex structured inputs. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 21–24. ACM (2018)
6. Calcagno, C., Distefano, D., O’Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. *J. ACM* **58**(6), 26:1–26:66 (2011)
7. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 71–80. ACM, New York (2008)
8. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, ICFP 2000, pp. 268–279. ACM, New York(2000)
9. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24730-2_15
10. Do, H., Elbaum, S., Rothermel, G.: Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact. *Empirical Softw. Eng.* **10**(4), 405–435 (2005)
11. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 416–419. ACM, New York (2011)
12. Galeotti, J.P., Rosner, N., López Pombo, C.G., Frias, M.F.: Analysis of invariants for efficient bounded verification. In: Proceedings of the 19th International Symposium on Software Testing and Analysis, ISSTA 2010, pp. 25–36. ACM, New York (2010)
13. Gligoric, M., Gvero, T., Jagannath, V., Khurshid, S., Kuncak, V., Marinov, D.: Test generation through programming in UDITA. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, pp. 225–234. ACM, New York (2010)
14. Goldberg, D.E.: *Genetic Algorithms in Search, Optimization and Machine Learning*, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston (1989)
15. Huang, W., Milanova, A., Dietl, W., Ernst, M.D.: Reim & ReImInfer: checking and inference of reference immutability and method purity. In: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA 2012, pp. 879–896. ACM, New York (2012)

16. Itzhaky, S., Bjørner, N., Reps, T., Sagiv, M., Thakur, A.: Property-directed shape analysis. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 35–51. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_3
17. Khalek, S.A., Yang, G., Zhang, L., Marinov, D., Khurshid, S.: TestEra: a tool for testing Java programs using alloy specifications. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, pp. 608–611. IEEE Computer Society, Washington, DC (2011)
18. Khurshid, S., Pasareanu, C.S., Visser, W.: Generalized symbolic execution for model checking and testing. In: Gavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 553–568. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36577-X_40
19. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **38**(1), 54–72 (2012)
20. Loncaric, C., Ernst, M.D., Torlak, E.: Generalized data structure synthesis. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, 27 May–03 June 2018, pp. 958–968 (2018)
21. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic testing of object-oriented software. In: van Leeuwen, J., Italiano, G.F., van der Hoek, W., Meinel, C., Sack, H., Plášil, F. (eds.) SOFSEM 2007. LNCS, vol. 4362, pp. 114–129. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-69507-3_9
22. Michalewicz, Z.: Genetic Algorithms + Data Structures = Evolution Programs, 3rd edn. Springer, Heidelberg (1996). <https://doi.org/10.1007/978-3-662-03315-9>
23. Nori, A.V., Rajamani, S.K., Tetali, S.D., Thakur, A.V.: The YOGI project: software property checking via static analysis and testing. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_17
24. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA 2007, pp. 815–816. ACM, New York (2007)
25. Ponzio, P., Aguirre, N., Frias, M.F., Visser, W.: Field-exhaustive testing. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, pp. 908–919. ACM, New York (2016)
26. Ponzio, P., Bengolea, V., Brida, S.G., Scilingo, G., Aguirre, N., Frias, M.: On the effect of object redundancy elimination in randomly testing collection classes. In: Proceedings of the 11th International Workshop on Search-Based Software Testing, SBST 2018, pp. 67–70. ACM, New York (2018)
27. Ponzio, P., Bengolea, V.S., Politano, M., Aguirre, N., Frias, M.F.: Replication package of the article: automatically identifying sufficient object builders from module APIs. <https://sites.google.com/view/objectbuildergeneration/>
28. Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of Java bytecode. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 179–180. ACM, New York (2010)
29. Rosner, N., Geldenhuys, J., Aguirre, N., Visser, W., Frias, M.F.: BLISS: improved symbolic execution by bounded lazy initialization with SAT support. *IEEE Trans. Softw. Eng.* **41**(7), 639–660 (2015)
30. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice Hall Press, Upper Saddle River (2009)
31. Sălciuanu, A., Rinard, M.: Purity and side effect analysis for Java programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 199–215. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-30579-8_14

32. Tillmann, N., De Halleux, J.: Pex–white box test generation for .NET. In: Beckert, B., Hähnle, R. (eds.) TAP 2008. LNCS, vol. 4966, pp. 134–153. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79124-9_10
33. Tillmann, N., de Halleux, J., Xie, T.: Parameterized unit testing: theory and practice. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010, pp. 483–484. ACM, New York (2010)
34. Visser, W., Mehrlitz, P.: Model checking programs with Java PathFinder. In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, p. 27. Springer, Heidelberg (2005). https://doi.org/10.1007/11537328_5
35. Visser, W., Păsăreanu, C.S., Pelánek, R.: Test input generation for Java containers using state matching. In: Proceedings of the 2006 International Symposium on Software Testing and Analysis, ISSTA 2006, pp. 37–48. ACM, New York (2006)
36. Website of the Apache Collections library. <https://commons.apache.org/proper/commons-collections/>
37. Website of the Java Genetic Algorithms Package. <http://jgap.sourceforge.net>
38. Website of the JGraphT library. <https://jgrapht.org/>

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter’s Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter’s Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.

