Constraint-Logic Object-Oriented Programming with Free Arrays of Reference-Typed Elements via Symbolic Aliasing

Hendrik Winkelmann[a] and Herbert Kuchen
Department of Information Systems, University of Münster, Leonardo-Campus 3, Münster, Germany

Keywords: Constraint-Logic Object-Oriented Programming, Constraint Solving, Symbolic Execution, Symbolic Arrays.

Abstract: Constraint-logic object-oriented programming is a young programming paradigm that aims to bring constraint-solving techniques to an audience more accustomed to imperative programming. A prototypical language of this paradigm, Muli, allows for the use not only of primitive-typed free variables, but also for free objects and free arrays of primitive-typed elements. In the work at hand, we extend the current version of Muli so that it supports free arrays of arrays and free arrays of objects. We do so by utilizing the concept of symbolic aliasing. Our evaluation shows that the presented approach can speed up program validation and test case generation, as well as solving complex constraint satisfaction problems.

1 INTRODUCTION

Constraint-Logic Object-Oriented Programming (CLOOP) is a programming paradigm combining the concepts of non-deterministic execution and constraint solving, known from Prolog (Wielemaker et al., 2012), with features known from object-oriented programming languages. These features comprise an imperative style of coding with mutability, dynamic dispatch, as well as inheritance (Dageförde and Kuchen, 2018; Dageförde et al., 2021). Instead of purely logic programming languages, CLOOP languages can be deemed more accessible for programmers that mainly use object-oriented languages and need to solve constraint(-logic) problems as parts of their applications. A particularly convincing example is the use of the prototypical CLOOP language Muli when applying these mechanisms to solve constraint problems for generating test cases (Winkelmann et al., 2022). One important feature of Muli is *free arrays*, i.e., unbound logic variables of an array type (Winkelmann et al., 2021). Up to now, the elements of free arrays were restricted to primitive types. Our main contribution in this paper is the extension of Muli's current runtime engine, Mulib (Winkelmann and Kuchen, 2022), with a mechanism for symbolically representing free arrays of primitive- as well as reference-typed elements while accounting for null values and mutability. The presented approach is general in that not only Satisfiability Modulo Theories (SMT) solvers such as Z3 (De Moura and Bjørner, 2008) but also Finite Domain (FD) solvers such as JaCoP (Kuchcinski, 2003) can employ it. We present use cases for free arrays in CLOOP and furthermore evaluate the approach with regards to its efficiency on several benchmark examples.

Subsequently, we first describe the core concepts of Muli and CLOOP, and motivate our contribution in Section 2. In Section 3 the implementation is described. In Section 4 we empirically compare the approach to an alternative and in Section 5, we contrast it from related work. We end the paper by concluding and outlining future work in Section 6. The overall implementation is available as open source software.[1]

2 AN OVERVIEW OF CLOOP

CLOOP is a paradigm that is based on *symbolic execution*, a technique well-known in the software validation and verification community (Cadar and Sen, 2013). Hence, it has been shown that CLOOP languages are a good fit for test case-generation (Winkelmann et al., 2022). Yet, they can also be used to represent search problems in an object-oriented manner.

[a] https://orcid.org/0000-0002-7208-7411

[1] https://github.com/NoItAll/mulib

Winkelmann, H. and Kuchen, H.
Constraint-Logic Object-Oriented Programming with Free Arrays of Reference-Typed Elements via Symbolic Aliasing.
DOI: 10.5220/0011849200003464
In *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2023)*, pages 412-419
ISBN: 978-989-758-647-7; ISSN: 2184-4895

2.1 Approach of Muli

CLOOP, as implemented by Muli (Dageförde and Kuchen, 2018), combines typical features of object-oriented programming languages with features such as non-deterministic execution and the native use of constraints by executing code encapsulated in so-called *search regions*. In these search regions, Muli allows to use *free* variables, i.e., variables with a symbolic value. If such a symbolic value is part of the evaluation of, e.g., the condition of an `if` statement (called if-condition in the sequel), a *choice point* with two *choice options* is created. The first option contains a *constraint*, as described by the mentioned condition. The other option contains the negation of the condition. Both cases are regarded separately.

The respective constraint is pushed to a *constraint stack*. An automated constraint solver, such as Z3 (De Moura and Bjørner, 2008), analyzes the satisfiability of the constraints on the constraint stack. If the constraints are satisfiable, the choice option is further evaluated, e.g., by entering the first branch of the `if` statement. The work at hand treats the improvement of free arrays in Muli.

2.2 Motivation for Free Arrays

Free arrays (Winkelmann et al., 2021) are a feature of Muli that allows for

1. accessing arrays with indices that are symbolic values,
2. initializing arrays with a symbolic length, and/or
3. initializing arrays that again have symbolic entries.

Free arrays in CLOOP can be used to express the *element* constraint, known from constraint programming, and combine it with mutability to express search problems in an elegant manner. Consider the following code depicting a search region in Muli:

```
1  public static int[] assign(
2          int[] mCaps, int[] workloads) {
3    int[] as = new int[workloads.length];
4    for (int i=0; i<workloads.length; i++) {
5      int indexOfM free;
6      int capOfM = mCaps[indexOfM];
7      if (workloads[i] > capOfM) {
8        throw Muli.fail(); }
9      mCaps[indexOfM] = capOfM-workloads[i];
10     as[i] = indexOfM; }
11   return as; }
```

In the search region, we look for an assignment of machines to workloads. The capacities of the machines, `mCaps`, as well as the set of workloads, `workloads`, are encoded as an array of integer values.

We assume that a machine finishes an assigned workload completely and not just a fraction of it. An assignment array, `as`, is initialized in line 3. Then, we iterate over all workloads to assign them (lines 4–10). For this, we first spawn a free index variable (line 5) and use it to select an entry of `mCaps` that we store in `capOfM` (line 6). The value `capOfM` represents the remaining capacity of some machine represented by `mCaps`. Thereafter, we create a choice point with two options (line 7). If the chosen machine has an insufficient capacity, a special exception is thrown declaring the choice option to be invalid (line 8). Thus, the runtime evaluates the choice option with the negation of the condition in line 7. If this constraint, in conjunction with constraints for earlier iterations that are already on the constraint stack, is satisfiable, we can assign the current workload to a machine: We store the difference of the machine's capacity and the current workload in the array (line 9). This is done to evaluate whether the chosen machine can still work on other workloads in later iterations. Finally, we store the index of the machine at the position of the current workload (line 10). There are two outcomes to this procedure. The first outcome is that, at some point, there is no way to make the negated if-condition `workloads[i] > capOfM` satisfiable. In this case, there are no valid paths to reach line 11 and Muli terminates without giving a solution to the user. The other outcome is that we reach line 11 and return the assignments. The solver is asked to assign a concrete value to each of the elements in `as`.

This simple example demonstrates the applicability of free arrays to a variety of scheduling and planning problems (Drexl and Kimms, 1997). The original implementation of Muli is already able to represent such problems (Winkelmann et al., 2021). However, it is not yet possible to represent reference typed-arrays, such as `int[][]` or `Machine[]` symbolically. While single `int` values are immutable, accounting for object identity, symbolic length, and mutability of the array elements is non-trivial. State-of-the-art tools (Păsăreanu and Rungta, 2010; Winkelmann et al., 2021) have left a symbolic representation for future work. Instead, simplifications are used that can lead to path explosion and can better be delegated to the constraint solver. Nevertheless, it is desirable to allow for a symbolic treatment of such arrays: Consider, for instance, an assignment problem where there are multiple periods and it is allowed to preproduce workloads of later periods. The machine capacities might then be represented via an `int[][]` array where the first dimension is the period in which the workload is to be worked on, while the second dimension is the capacity of a machine in this period. Moreover, in the context of object-oriented programming, it might be even more desirable to represent machine capacities as an array of `Machine`s, exhibiting specific behavior using dynamic polymorphism (Dageförde et al., 2021). Another reason is that, in the context of symbolic execution, built-in procedures for dealing with array constraints oftentimes are not as performant as custom-tailored ones (Perry et al., 2017). In the following section, we describe a solver-independent high-level approach for treating array constraints. We then extend this approach to allow for arrays of reference types while specifically accounting for null-values, arrays of arrays, and arrays of other object-types. For space reasons, we will assume arrays with a concrete length in the subsequent elaborations.

3 APPROACH & IMPLEMENTATION

In a Muli program, some variables and data must be represented in a constraint solver. For instance, when we have a constraint such as $i < 0$, we must create a solver-specific representation of i, 0, and the constraint $i < 0$ to push to the constraint stack of the solver. For primitive data types and operations, this is rather straight-forward. In the following, we describe how we represent objects and/or their *content*. We follow the Java denomination where arrays are objects and thus have an object identity. *Content* is used to either describe the elements in the case of an array, or, in the case of a non-array object, its field values. Note that we deliberately use the phrase *representing an object **for** the solver*. This is because the object itself is not represented in some solver-specific fashion, e.g., via algebraic data types or an array theory oftentimes supported by SMT-type constraint solvers (Barrett et al., 2017). Instead, we create a representation that transforms accesses to the object to constraints, which then are forwarded to the solver.

3.1 Representing Selects and Stores

For representing operations on an integer array a we must support two operations: Selecting from a with index i and storing an element e in a with index i.

Algorithm 1 depicts the devised high-level procedure (compare with Perry et al. (2017)) where the goal is to encode a select operation $a[index]$ as a constraint. This constraint is then added to the constraint stack to assure that the selected value, here passed as *Sprimitive val* (line 4), is a valid element of the respective array at *index* (line 3). An instance of *Sprimitive* represents either a concrete primitive value or a symbolic value. An instance of its subtype, *Sint*, represents either a concrete integer value, or a symbolic one (Winkelmann and Kuchen, 2022).

Algorithm 1: Simplified procedure for encoding a select constraint.

```
1  select (ArrayHistory ah,
2          Constraint selectGuard,
3          Sint index,
4          Sprimitive val) : Constraint
5      List[(Sint,Sprimitive)] initial =
6          ah.getInitialState ();
7      Constraint result = true;
8      foreach (i,v) ∈ initial do
9          Constraint implication =
10             $(index == i) \implies (val == v)$;
11         $result = result \land implication$;
12     List[(Constraint,Sint,Sprimitive)] stores =
13         ah.getStores ();
14     foreach (c,i,v) ∈ stores do
15         Constraint validStore =
16             $(c \land index == i)$;
17         Constraint implication =
18             $(validStore \implies (val == v))$;
19         result = implication
20             $\land(\neg(validStore) \implies result)$;
21     return $selectGuard \implies result$;
```

To better understand this algorithm, consider the following example with an integer array a:

```
int[] a = new int[] {4, s};
int index free;
int val0 = a[index];
```

We create an array a, select from it using a free index *index*, and store the value in *val*0. This access can effectively be encoded by the following constraint:

$$index \geq 0 \land index < 2 \land (index = 0 \implies val0 = 4)$$
$$\land (index = 1 \implies val0 = s)$$

The constraint $index \geq 0 \land index < 2$ is necessary to ascertain that *index* is in the valid index range of a. In the following, we abstract from this length-constraint. This condition, together with the following two implications, fix *val*0 to be either 4 or s without prematurely assigning a fixed value to *index* or *val*.

Effectively, this constraint is generated by executing lines 5-11 in Algorithm 1. *ArrayHistory* here is a well-known representation of the respective array (Perry et al., 2017) containing two lists. The first list, the one that is used in lines 5–11, contains the index-value pairs of the array at the time it was represented for the solver. Thus the initial set of index-value pairs is retrieved (lines 5 and 6) and implications are created and conjoined (lines 7–11). Each time *index* equals one of the indices i, the selected value *val* must equal the value v (lines 9 and 10). *index* has been restricted to a valid range before entering Algorithm 1.

The second list, which is empty in this example, contains triples of store statements executed for the array. As an example for stores, again consider the array a and the following operations performed on it:

```
int i0 free; int i1 free;
a[i0] = 5;
int val1 = a[i1];
```

In this example, we store 5 in a using the free index $i0$. The subsequent select operation using the free index $i1$ must take into account that some value in a has been overwritten. To achieve this, the store adds a triple to the aforementioned second list of $ArrayHistory$. For now, we assume the first element of the triple as well as $selectGuard$ to equal $true$. The second and third element of the triple again form a key-value pair. In the example, the triple $(true, i0, 5)$ is added to the second list of $ArrayHistory$. The subsequent select operation `a[i1]` now effectively generates the following constraint:

$$(i0 = i1 \implies val1 = 5) \land (i0 \neq i1 \implies ((i1 = 0 \implies val1 = 4) \land (i1 = 1 \implies val1 = s)))$$

$(i0 = i1 \implies val1 = 5)$ is an implication using the stored index and value. The second part of this constraint, $(i0 \neq i1 \implies ...)$, is the constraint in the form known from the previous scenario *guarded* by the condition that the index used for selecting, `i1`, does not equal the index used for storing, `i0`. Algorithm 1 implements the effect of the store statement in lines 12–20. We encode overwriting a value via an implication. If the antecedent of the implication (lines 15 and 16) holds, *val* must equal v of this store (lines 17 and 18). Otherwise, $validStore$ must not hold, and the non-overwritten prior key-value pairs must be considered (line 20). Finally, we return the overall constraint (line 21).

3.2 Representing Non-Aliasing Objects

When extending the representation approach from Subsection 3.1 to arrays of reference types and, in general, objects, it must be considered how we represent object identity symbolically. For this, we endow each object with an additional integer variable of type `Sint`. These integer variables represent the identity of an object and subsequently are called *(object) identifiers*. If an object identifier equals another object identifier, these two objects are considered to be aliases of one another.

Each object created by 'usual' means, i.e., in Java by using the `new` operator, receives a unique concrete integer number as its identifier. In consequence, no two objects that are created in this fashion can be aliases of one another.

We represent the elements of an array a as follows: If a is an array with primitive-typed elements, such as an `int[]`, the initial elements of a are represented as a list. Values that are stored in the array are added to the aforementioned list of stores to account for symbolically overwriting the initial values. Objects are represented via maps. The keys of the map are the names of the object's fields. If a field is primitive-typed, we directly store its value at the time of representation. On the other hand, if the field's type is a reference type, we instead store the **identifier** of the referenced object in it. Analogously, arrays with reference-typed elements are represented by storing the list of identifiers of the elements in a. Nulls are represented by the reserved identifier -1.

As an example, consider the following code snippet, creating an array with reference-typed elements:

```
T o0 = new T();
U u = new U(); u.f2 = 42.0;
o0.f0 = 7; o0.f1 = u;
T o1 = new T();
o1.f0 = 5;
T[] a = new T[] {o0, o1};
```

Here, we create an array `a` with two elements, `o0` and `o1`. Their fields `f0` and `f1` are set to suitable values. If array `a` were to be represented for the solver, the following representations, here depicted in JSON, are the outcome:

$Ro_0 = \{"id" : 0, "es" : \{"f0" : [[7], []], "f1" : [[1], []]\}\}$
$Ru = \{"id" : 1, "es" : \{"f2" : [[42.0], []]\}\}$
$Ro_1 = \{"id" : 2, "es" : \{"f0" : [[5], []], "f1" : [[-1], []]\}\}$
$Ra = \{"id" : 3, "len" : 2, "es" : [[0, 2], []]\}$

The representation of `o0`, Ro_0, consists of a concrete identifier, here 0, and its fields *es*. The representation for each field consists of a pair of lists. The first list represents the initial value of the field. The second element of the pair is the aforementioned list of store triples for when the field is set to a new value. For `o0.f1`, we store the identifier of *Ru*, here, 1. For `o1`, we proceed similary, except that `o1.f1` is `null`, and thus the identifier -1 is stored. Finally, the representation of the array `a`, *Ra*, contains an identifier, and the length of `a`, len. Moreover, it contains the identifiers of its represented elements in the same data structure that is used for arrays with primitive typed-elements.

Consider Algorithm 2 in which we describe how we can retrieve the field value from the representation for the solver of a non-array object. Since we only represent a single field value, *index* of Algorithm 1 is always set to 0. In turn, setting a new value adds a new triple $(guard, 0, val)$ to the list of stores of the array history. For array objects, each representation only has one $ArrayHistory$ and *index* is not always 0.

Algorithm 2: Simplified procedure for getting a field's value.

```
1 getField(Constraint guard,
2          String fieldName,
3          Sprimitive val) : Constraint
4     ArrayHistory ah =
5         this.fields.get (fieldName);
6     return select (ah, guard, 0, val);
```

3.3 Representing Aliasing Objects

It becomes necessary to represent an object for the constraint solver if it is

1. an array that is accessed with a free index for the first time,
2. contained in an object that is in the process of being represented for the solver, or if it is
3. stored into an object that is already represented for the solver.

Such an object, or its content, is a potential target for *symbolic aliasing*. In the following, we say that we return an *aliasing object* where the elements of the array are *aliasing targets* thereof. To account for aliasing, while running the program, each time the content of an aliasing object or an aliasing target is to be accessed, instead its representation for the constraint solver is accessed.

Aliasing objects are created either when

1. an array of reference-typed elements is selected from with an unknown index,
2. an aliasing array of reference-typed elements is selected from, or
3. an object is retrieved from a reference-typed field of a non-array aliasing object.

The identifier of aliasing objects is a symbolic integer. Consider an array `a = {o0, o1}` of type `T[]`, where T is a reference type. Every time `a` is accessed via an unknown symbolic index i, we spawn an object, o_i, of type `T`. The identifier of o_i is a symbolic integer value. The domain of $o_i.id$ is determined by selecting from the representation of `a`. Effectively, the following constraint is pushed onto the constraint stack:

$$((i=0) \implies (o_i.id = o_0.id)) \\ \wedge((i=1) \implies (o_i.id = o_1.id))$$

In consequence, $o_i.id$ equals either $o_0.id$ or $o_1.id$. In other words: o_i is an aliasing object and o_0 and o_1 are aliasing targets.

Aliasing objects are immediately represented for the solver. When representing the aliasing object, we generate a metadata constraint. The metadata constraint enforces that the object can only be null if one of its aliasing targets can be null, and, in case of an array, that the length is equal to the length of one of its aliasing targets. It does so by restricting the identifier of the aliasing object to be in the set of identifiers of the aliasing targets, and implies the respective state using the symbolic identifier identifier.

After setting its identifier and pushing the metadata constraint, the aliasing object is returned to the search region.

If an instance method is called or the content of o_i is accessed, it is checked if the identifier $o_i.id$ can equal -1. If this is the case, we create a choice point. The choice options are that either a null-pointer exception is thrown or that we continue assuming that $o_i.id \neq -1$.

o_i itself does not have any content of its own: Again, consider array a of type $T[]$. Here, the selected aliasing object o_i is of type T. If we get a value *val* from the field $o_i.f$, Algorithm 2 is executed on o_0 and o_1. Additionally, we add *guards* based on the o_i's symbolic identifier to these calls, effectively generating the following constraint:

$$(Ro_0.getField(o_i.id = o_0.id, f, val)) \\ \wedge(Ro_1.getField(o_i.id = o_1.id, f, val))$$

Here, Ro_0 and Ro_1 are the respective representations of o_0 and o_1 for the constraint solver. Note that we pass the constraint $o_i.id = o_0.id$ or $o_i.id = o_1.id$ to the *guard* parameter of Algorithm 2. By using this guard, and since we restricted $o_i.id$ to be in $\{o_0.id, o_1.id\}$, the result is a correct select constraint given that o_i is an alias of o_0 or o_1. The constraint-generating procedure is described in more detail in Algorithm 3. The algorithms for selecting and storing in arrays or setting a field value for an aliasing object behave analogously. In it, we iterate over the set of representations of aliasing targets (lines 5–12). Since we access the aliasing target's `getField`-method, we must add a guard for showing that this is a conditional select statement (lines 7 and 10). Since this guard is used in Algorithm 1, line 21, as the argument to *selectGuard*, it is stated that *val* is only restricted by the content of o_0 or o_1, if *this* is an alias of the current aliasing target *a*. It may even occur that the current aliasing object is the aliasing target of another aliasing object. Thus, we have to conjoin this guard with the guard constraint passed via the parameter *guard* of Algorithm 3 (line 8). Thereafter, we call `getField` on the aliasing target *a* (lines 9–11). We conjoin the outcome of this method call to the result (line 12). Finally, we return the resulting constraint (line 13).

On the other hand, setting a new value for the content of an aliasing object modifies each aliasing target's representation. Recall that the triple has the form (c, i, v) where i and v are the index and value

Algorithm 3: Simplified procedure for getting an aliasing non-array object's field value.

```
getField(Constraint guard,
            String fieldName,
            Sprimitive val) : Constraint
    Constraint result = true;
    Set[NonArrayRep] at = this.aliasingTarget;
    foreach $a \in at$ do
        Constraint idEq = a.id == this.id;
        Constraint guards = $idEq \wedge guard$;
        Constraint gf =
            a.getField(guards, a.fields,
                fieldName, val);
        result = $result \wedge gf$;
    return result;
```

of the store statement. c is the *guard* statement using the identifier of the aliasing object and the identifier of the aliasing target. In other words, storing a value in an aliasing object o_i also is a *conditional* store for an aliasing target, e.g., o_0, based on the constraint $o_i.id = o_0.id$. The validity of a store statement is considered in Algorithm 1 in lines 15–18, where the added guard statement is used to evaluate whether the store is valid for the current select call. Thus, by doing so we also can reuse Algorithm 1 for representing conditional field updates.

4 EVALUATION

In the following, we will compare the symbolic approach described in this paper with an *eager* approach. The eager approach works as follows: As soon as an array of objects, for instance an `int[][]`, is selected from or stored within using a symbolic index, choice options are created. Each choice option is the choice of a concrete value for the symbolic index. Internally, for each of these choice options subsequent operations are considered to be on disjoint paths, i.e., the eager approach creates one path for each chosen value for such array accesses. In summary, in the eager approach we *eagerly* decide on an index value for a given array operation while in the symbolic approach proposed in this paper, we delegate this choice to the constraint solver using the mechanisms described in Section 3. In consequence, the need to create the mentioned paths is removed for such symbolic array accesses. In the past, approaches similar to the eager approach have been used in symbolic execution tools such as Symbolic Pathfinder (Păsăreanu and Rungta, 2010).

For evaluating symbolic index accesses, we have devised six scenarios of varying complexity, where each one represents an assignment problem. We executed all scenarios on an HP ProBook 445 G7 laptop with an AMD Ryzen 5 4500U CPU and 16 GB of working memory. Each scenario has been executed 20 times. We then removed the first five measurements to assure that the first few iterations, oftentimes exhibiting a more variable run time profile, do not skew the results. A time budget is set so that after 30 seconds no new explorations of the search region are started.

The first scenario, `PCAP`, is the initial motivational example from Subsection 2.2 executed with 18 machines and 24 workloads. `MPCAP` is a scenario in which we have two periods. There are a total of nine machines working on 12 workloads in each of the two periods. Machines are represented as an `int[][]`, where the first dimension determines the period in which a machine works and the second dimension determines the concrete machine. `MCAP` is the same as `PCAP`, only that we now encode the machines as objects, i.e., we use `Machine[]` instead of `int[]`, where `Machine` has a single `int`-field. Hence, instead of `int` values, we retrieve `Machine` objects and access their fields. This scenario serves the purpose of evaluating the computational cost associated with introducing a layer of indirection, thus making use of symbolic aliasing (see Subsection 3.3). On the other hand, `MPMCAP` analogously uses an array of type `Machine[][]` instead of an `int[][]` argument. `MCAP-RED` defines a smaller variant of `MCAP` with eight machines and eleven workloads. Finally, `DLSPV` denotes a variant in which certain workloads can only be worked on by certain machines and each machine can only work on one single workload in a given period, i.e., it is a simplification of the *Discrete lot-sizing and scheduling problem* (Drexl and Kimms, 1997). This example contains nine machines and four periods, each with nine workloads. Z3 (De Moura and Bjørner, 2008) is used as an incremental constraint solver to evaluate the valid paths through the program.

We evaluate multiple configurations for solving the scenarios. The results are given in Table 1. For each configuration we specify the mean time needed to find a first path solution as well as the standard deviation in the first row, and the mean time to extract further paths until there are no more valid paths and its standard deviation in the second row of a cell. The next column shows the total number of found path solutions, followed by the number of explicit *fails* (see line 8 in the example of Subsection 2.2).

For the `PCAP` scenario, we compare the high-level procedure described in Algorithm 1 (`HL`) with an implementation using the array theory of Z3 (`Solver-specific`). Compared to the Z3 implementation our high-level approach is able to retrieve a first path solution and validate that there are no more paths in the search region considerably faster. Furthermore, the standard deviation here is lower than for the Z3 implementation. Z3 employs heuristics that result in comparatively high, yet reproducible, differences in the various run times, causing a high standard deviation in both cases.

In the `MPCAP` scenario we compare our purely symbolic approach, described in Section 3, with the aforementioned eager approach. After retrieving an array of type `int[]` from the set of machines, represented as an `int[][]`, the arrays with primitive-typed elements are treated with the high-level array theory. As can be seen, for `MPCAP`, the eager approach times out and does not find any path solution. This can be attributed to a high number of costly constraint solver calls where the constraint stack is unsatisfiable. In contrast, the purely symbolic approach can solve the problem and extract a solution in about one second.

Table 1: The mean run times and their standard deviation for finding a first and for enumerating all path solutions, as well as the mean number of found path solutions and fails for the benchmark examples. The times are given in seconds rounded to two digits, the number of path solutions and fails are rounded to a full number. TO denotes a time out for all runs.

Scenario	Setting	Time	#PS+#Fails
PCAP	Solver -specific	10.83 (4.22)	1+23
		11.35 (4.28)	
	HL	2.90 (2.00)	1+23
		3.04 (2.02)	
MPCAP	Eager	TO	0+5
	Symbolic	0.92 (0.21)	1+35
		1.18 (0.26)	
MCAP	Eager	TO	0+266572
		TO	
	Symbolic	4.16 (3.97)*	1+23
		4.28 (3.97)	
MPMCAP	Eager	18.67 (0.07)	1+217550
		TO	
	Symbolic	0.43 (0.09)	1+32
		0.50 (0.11)	
MCAP -RED	Eager	4.60 (0.02)	144+333752
		TO	
	Symbolic	0.05 (0.02)	1+10
		0.06 (0.02)	
DLSPV	Eager	2.40 (0.01)	1+12082
		2.48 (0.01)	
	Symbolic	0.30 (0.03)	1+96
		0.36 (0.04)	

The `MCAP` scenario was designed to evaluate the cost of the additional identifier *guard* constraints. Since we use free indices for selecting from an object, i.e., we select from `Machine[]`, the eager approach here is forced to perform a brute-force search and times out. Hundreds of thousands of fails were encountered without finding a single path solution. It must be noted that for `Symbolic` one of the fifteen measurements timed out. The depicted times do not consider this time out. For the eager configuration, `MPMCAP` enforces a brute-force approach: When comparing the number of paths, one can see that the eager approach suffers from path explosion and takes much longer to find its first solution.

Finally, `DLSPV` shows a scenario in which there is a reasonable number of overall paths, so that no approach times out. The symbolic approach outperforms the eager approach. Compared to the eager approach, the symbolic approach has to regard far fewer paths, as it *summarizes* the decision on an index.

5 RELATED WORK

The use of free arrays in CLOOP is novel with the exception of primitive free arrays using the array theory available in some SMT solvers (Winkelmann et al., 2021). There are, however, related subjects in the area of constraint programming and software validation.

In the area of constraint programming, tools (Kuchcinski, 2003) offer the *element* constraint. By means of the element constraint, it can be checked whether a certain value is located in a list in a declarative manner. This resembles selecting from an array with a free index. However, since constraint programming is declarative, it does not directly account for mutating objects or overwriting values in the list. Also, Muli allows for defining custom data structures instead of using integer encodings.

Similarly, approaches like *JSetL* (Rossi and Bergenti, 2015) offer non-deterministic programming with logical variables in the form of a library, making use of a custom API and focus on lists and sets. For instance, to add a new choice point, the `Solver`-API must be called. In contrast, Muli allows for mutability and uses native language constructs such as if-conditions to automatically create choice points where a free variable is involved. Furthermore, via the contributions of this paper, Muli enables the use of objects of user-defined classes in arrays which can also represent set operations.

Muli can be used to generate test cases (Winkelmann et al., 2022). As can be seen in Section 4, using our approach for symbolic arrays can significantly reduce the number of encountered paths. This is in line with research for *path merging* conducted, for instance, by *Java Ranger* (Sharma et al., 2020), where, e.g., `if-then-else`-constructs are summarized and the differentiation of various paths through the program is delegated to the constraint solver. In consequence, more complex constraints are pushed to the constraint stack, yet, fewer paths have to be evaluated by the tool. While promising, *Java Ranger* does not treat purely symbolic arrays (Sharma et al., 2020).

Our approach draws from past research on speeding up array constraints (Perry et al., 2017). This research is comparable to Algorithm 1. Our implementation differs in that we use implication opposed to nested `if-then-else` statements with summarized index ranges as the latter has reduced the performance for our benchmark examples. Furthermore, we provide a mechanism to express symbolic aliasing, thus enabling a purely symbolic treatment of arrays of objects for all solvers supporting negation, disjunction, and conjunction of constraints as well as value equality.

6 CONCLUSION AND FUTURE WORK

We have described an approach for symbolically representing arrays of arrays and, more generally, arrays of objects. We have shown that the approach outperforms related work for retrieving path solutions. In the future, it might be interesting to compare the performance of other types of constraint solvers, such as FD solvers, with SMT solvers in the context of CLOOP. This is possible since the presented mechanisms pose very few requirements towards employed solvers.

REFERENCES

Barrett, C., Fontaine, P., and Tinelli, C. (2017). The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90.

Dageförde, J. C. and Kuchen, H. (2018). A constraint-logic object-oriented language. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 1185–1194, New York, NY, USA. Association for Computing Machinery.

Dageförde, J. C., Winkelmann, H., and Kuchen, H. (2021). Free objects in constraint-logic object-oriented programming. In *23rd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2021, New York, NY, USA. Association for Computing Machinery.

De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg. Springer-Verlag.

Drexl, A. and Kimms, A. (1997). Lot sizing and scheduling — survey and extensions. *European Journal of Operational Research*, 99(2):221–235.

Kuchcinski, K. (2003). Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems*, 8(3):355–383.

Păsăreanu, C. S. and Rungta, N. (2010). Symbolic pathfinder: Symbolic execution of java bytecode. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, page 179–180, New York, NY, USA. Association for Computing Machinery.

Perry, D. M., Mattavelli, A., Zhang, X., and Cadar, C. (2017). Accelerating array constraints in symbolic execution. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, page 68–78, New York, NY, USA. Association for Computing Machinery.

Rossi, G. and Bergenti, F. (2015). Nondeterministic programming in java with jsetl. *Fundamenta Informaticae*, 140(3-4):393–412.

Sharma, V., Hussein, S., Whalen, M. W., McCamant, S., and Visser, W. (2020). Java ranger: Statically summarizing regions for efficient symbolic execution of java. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 123–134, New York, NY, USA. Association for Computing Machinery.

Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T. (2012). Swi-prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96.

Winkelmann, H., Dageförde, J. C., and Kuchen, H. (2021). Constraint-logic object-oriented programming with free arrays. In Hanus, M. and Sacerdoti Coen, C., editors, *Functional and Constraint Logic Programming*, pages 129–144, Cham. Springer International Publishing.

Winkelmann, H. and Kuchen, H. (2022). Constraint-logic object-oriented programming on the java virtual machine. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1258–1267, New York, NY, USA. Association for Computing Machinery.

Winkelmann, H., Troost, L., and Kuchen, H. (2022). Constraint-logic object-oriented programming for test case generation. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, SAC '22, page 1499–1508, New York, NY, USA. Association for Computing Machinery.