Working Papers

ERCIS – European Research Center for Information Systems
Editors: J. Becker, J. vom Brocke, T. Brandt, F. Gieseke, B. Hellingrath, T. Hoeren, S. Klein, H. Kuchen, J. Varghese

Working Paper No. 39

An Efficient Implementation of a Runtime for Constraint-Logic Object-Oriented Programming

Hendrik Winkelmann

ISSN 1614-7448

cite as: Hendrik Winkelmann: An Efficient Implementation of a Runtime for Constraint-Logic Object-Oriented Programming. In: Working Papers, European Research Center for Information Systems No. 39. Eds.: Becker, J. et al. Münster 2024.

Contents

List of Figures

List of Tables

1 Introduction

In this technical report the symbolic execution engine 'Mulib' will be outlined in its architecture, main algorithms, and design decisions. Mulib employs a rather new approach of program transformation (see Section 2) and thus, technical details on the implementation of such a system are thought to be insightful. Mulib first and foremost implements the runtime semantics of the Constraint-Logic Object-Oriented Programming (CLOOP) language Muli [18]. Muli employs symbolic execution [11] as a technique to, e.g., solve constraint-satisfaction problems and generate test cases [19, 55].

1.1 An Exemplary Program

Consider the following code for an example of how Muli can be utilized to solve a decision variant of the well-known knapsack problem. The goal here is to load the knapsack with items so that the capacity of the knapsack is fully utilized. Since Muli employs concepts of logic programming, it strives to return all such knapsacks, i.e., the call to the displayed method can potentially output multiple (or no) results.

```
1  static ArrayList<Item> fillKnapsack(){
2     final int capacity = 42;
3     List<Item> items = getPossibleItems();
4     int weight = 0;
5     ArrayList<Item> result = new ArrayList<>();
6     for (Item item : items) {
7        if (weight == capacity) break;
8        boolean take free;
9        if (weight + item.weight <= capacity && take) {
10          result.add(item);
11          weight += item.weight;
12       }
13    }
14    assume(weight == capacity);
15    return result;
16 }
```

In a first step, the capacity is set to some (arbitrary) value and a list of `Item`s is retrieved (lines 2 and 3). Then, the weight of the knapsack is initialized to `0`, meaning that currently, no `Item`s have been loaded. Thereafter, the list in which the loaded `Item`s are stored is initialized and each of the `Item`s is iterated over (lines 5–13). It is checked whether the initialized weight already equals the capacity (line 7). Only if this is not the case, a new item is tried to be loaded. A new `boolean` is initialized in a special way (line 8). Muli modifies the Java syntax and introduces the `free` keyword [18]. A variable that is initialized using the `free` keyword contains a *symbolic value*, i.e., a value which is not yet restricted to one fixed value. Here, `take` is either `true` **or** `false`. Next, it is evaluated whether the currently regarded `item` shall be loaded into the result list (lines 9–12). This is done using an `if` statement. Only if the sum of the item's weight and the currently loaded weight does not exceed the capacity, the item is loaded. Furthermore, it is also desirable to evaluate the case where `item` is not loaded. To model this, the `if` statement also depends on the variable `take` that contains a symbolic value. If a symbolic value becomes part of the evaluation of an `if` statement, the execution creates two forks. These forks represent nondeterminism in the execution of the program: Depending on the *label* of the symbolic value, i.e., a concrete value for it, the program executes different paths. In the first fork, it is checked whether the condition can hold, i.e., `take` is `true`. In the second fork, the negation of this condition is checked. If a constraint solver determines that the new condition, conjoined with previous conditions, is satisfiable, the

`then`-branch is executed.[1] In the given example, `item` is added to the result list of `Item`s and `item`'s weight is added to the total weight (lines 10 and 11). At some point determined by a search strategy, the runtime restores the program's state to the state when aforementioned forks was first encountered. The remainder of the program is then executed using the negation of the condition. In the given example, due to the loop, the execution will evaluate the next `Item` (line 6). After considering all items (or potentially breaking prematurely due to line 7), it is checked whether the weight equals the capacity (line 14). If this is not the case, a special exception is thrown that discards the current fork of execution. In consequence, only if `weight == capacity`, a result is returned. If there is a solution to this problem, a list of `Item`s is returned.

1.2 Scope of This Technical Report

Mulib [52, 53] replaces the old runtime environment of Muli that is based on a custom Java Virtual Machine (JVM) and a custom compiler [36, 17]. In the following, when referring to 'Muli', the programming language is meant. In contrast Muli$_{sjvm}$ denotes the first implementation of Muli using a custom JVM and a custom compiler [17]. Mulib denotes the new runtime environment for Muli.

In the following elaborations, it is assumed that the reader has some familiarity with the stack-based Java bytecode [33], CLOOP, and symbolic execution in general. This document will focus on the Java programming language and tools and concepts related to it. It is suggested that readers that are not familiar with symbolic execution first read appropriate fundamental literature (such as Cadar and Sen [11] and Khurshid, Păsăreanu, and Visser [29]).

Moreover, there are several publications on Muli available [18, 21, 17, 19, 55, 51, 22, 52, 53]. In contrast to the publications on Mulib [52, 53], this technical report is meant as an extension to convey technical details and the software engineering design decisions. Even though the various concepts are introduced to some degree, at least the core of CLOOP and Muli should be known (see, e.g., Dageförde and Kuchen [19] for applications of CLOOP, or Majchrzak and Kuchen [36], Dageförde and Kuchen [18], and Dageförde and Kuchen [17]).

This document is structured as follows: First, in Section 2, we mention related work and approaches for enabling the symbolic execution of (Java) programs. Mulib's design is contrasted from these already existing tools and we outline the advantages of the new design. Section 3 gives an overview of how users can interact with Mulib and what the workflow and general architecture looks like. Thereafter, in Section 4 the type system of Mulib is outlined that is used to enable symbolic execution. In turn, Section 5 provides a description of several core algorithms used to transform programs to programs using the type system of Mulib so that the program can be executed symbolically. In Section 6, we outline the framework by means of which search is conducted in a transformed program. During search, Mulib uses a constraint solver. Thus, Section 7 shows how constraint solvers can be integrated into Mulib. Section 8 describes some other noteworthy features of Mulib. In turn, Section 9 describes possibilities for future work on Mulib while Section 10 draws a conclusion.

Mulib, alongside a comprehensive documentation, is open source.[2].

[1] This is the behavior of symbolic execution [11].

[2] `https://github.com/NoItAll/mulib`

2 Related Work

Broadly speaking, most tools for symbolic execution can be subdivided into three types of approaches. Since each approach can potentially be used to implement Muli, their advantages and disadvantages should be regarded. In this section these approaches are summarized and the best fit for creating the execution engine Mulib is identified.

First, symbolic execution tools can use an interpreter or a custom virtual machine for symbolic instructions. As this technical report focusses on Java, in the following, both, interpreters as well as custom symbolic virtual machines, will be subsumed using the term Symbolic Java Virtual Machine (SJVM).

In these SJVMs, certain bitcode or bytecode instructions are extended to account for the semantics of symbolic execution. For instance, the operation of adding two integer values might not have concrete integer values as its operands. Instead, symbolic expressions might be the operands. As another example, conditional jumps potentially yield a choice point with two choice options, i.e., two forks of executon, if the condition involves symbolic values. Exemplary tools comprise Muggl [36] as the underlying structure of Muli$_{sjvm}$, Symbolic PathFinder [29], and KLEE [10] (for LLVM bitcode).

A downside of this approach is its inefficiency [38]: Every single bytecode instruction is interpreted and thus can hardly benefit from either static compilation or just-in-time compilation [15]. Furthermore, the runtime is quite complex and developers of new features oftentimes must deal with internals such as stack frames, a program counter, low-level representations of object references and hard-to-debug side-effects.

The second, more light-weight, approach is the instrumentation of the program. Instead of executing the program using an SJVM, an extended program is executed using the original runtime, e.g., C code is executed in a binary format and Java code is executed on a standard JVM. This extended program still has the semantics of the original program but has additional *listening* operations, the instrumentation, tracking the state of the program. The program is executed concretely, just as it would be executed without instrumentation. Additionally, the program's execution is listened to using the instrumentation operations, and the concrete values of the program are mapped to a symbolic representation. For instance, in COASTAL [48], a concolic executor for Java, instructions such as `VM.methodInsn(int instr, int opcode, String owner, ...)` [49] are inserted after bytecode instructions (such as, in the given example, the invocation of a method) and trigger side-effects to map concrete operations and values to potentially symbolic counterparts. Exemplary tools comprise COASTAL [48], GDart [37] and its predecessor JDart [35].[3]

The upside of this approach is that the performance of the native runtime is largely preserved. The overhead is reduced to calls to the constraint solver and maintaining a symbolic program representation to and from which the concrete program representation is mapped. In other words: Two sets of values are maintained, namely concrete values and symbolic values. The concrete values are *labels* of the symbolic values, i.e., they satisfy all constraints associated with their respective symbolic value. Such tools employ *concolic* execution [11]. The concrete execution drives the symbolic one, i.e., whatever path condition is satisfied by the concrete program state is the path the concolic execution tool follows. Since the values of the concrete program state thus are always valid with regards to the chosen path, several satisfiability checks are obsolete; - the constraint solver does not have to be invoked to verify whether the taken path is valid. As long as constraints are added to the program using choice points, the labels are always valid with regards to one of the two choice options. However, a downside becomes apparent if choice points are not the only source of constraints. To understand this important difference, consider the following two pseudocode snippets:

[3] GDart uses built-in runtime tooling to achieve an instrumentation.

```
int i0 free;
int i1 free;
if (i0+i1 > 3) {
    ...
} else {
    ...
}
```

```
int i0 free;
int i1 free;
assume(i0+i1 > 3);
```

In both listings, two symbolic variables `i0` and `i1` are created. Concolic execution will assign a *seed* label value to them, for instance, `0`. Thus, when first encountering the case distinction in the code snippet on the left-hand side, it might be determined that the `else`-branch is traversed first. Thereafter, it is checked whether inputs for accessing the `then`-branch can be generated. In this instance, concolic execution works as intended: Since `i0` and `i1` are seeded with valid values, either the `then` or the `else` branch is satisfiable.

On the other hand, the code on the right-hand side treats the case where we *assume*, i.e., add a constraint, that the sum of `i0` and `i1` is larger than `3`. `i0` and `i1` again are seeded with some value, e.g., `0`. However, in this case, the pushed constraint is not satisfied by the concrete state. In the best case, the concolic executor notices that the current labels are stale and must generate new inputs. The underlying problem is that the labels were formed prematurely. Only a pure symbolic execution can account for *assume*-type operations, or constraints that are added without creating a choice point in general, as the labels are only generated at the very end of the execution. The generation of new inputs, in turn, is a costly operation and hence, the execution becomes inefficient. While such *assume*-operations are not expected to occur often in typical symbolic/-concolic execution for test case generation, they should be accounted for in CLOOP: For solving constraint satisfaction problems, the programmer might decide to restrict the domains of symbolic values without wanting to create a choice point. Additionally, *assume* also has benefits in pruning the search space in symbolic execution (see, for instance, Subsubsection 8.2.4 Enumeration Item 2 where a constraint is pushed assuming that an index is in the range of an array) and should thus be accounted for. Another downside of the approach is that still, the overall system is vastly dependent on side-effects. The concrete execution must always be mapped to the symbolic state, creating a layer of indirection and increasing debugging-efforts.

The third type of approach has found very little recognition in literature and is employed by LART [31] that evolved from a symbolic execution extension of the model checker DIVINE [32]. In a first step, a program transformation is applied to the program. The new program directly contains instructions for symbolic execution. Thereafter, this code can be interpreted or executed. This is the approach that Mulib chooses to implement.

Instead of executing a program with a virtual machine or instrumenting the original program, the program is transformed into a format that directly accounts for symbolic execution. For instance, since the primitive types of Java do not account for symbolic expressions, `int` fields and variables, potentially holding a symbolic value, are transformed to the type `Sint` [52]. `Sint` is a pre-implemented library class that has subclasses accounting for concrete values and for symbolic values. Its subclasses, e.g., `ConcSint` and `SymSint` either have a concrete `int` value or a Directed Acyclic Graph (DAG), representing a (mathematical) expression. Many of such classes, including `Sdouble`, `Slong`, `Sbyte`, `Sbool`, `Schar`, `SintSarray` (for `int[]`), etc., have been implemented. The underlying execution framework can compute, e.g., the addition of a concrete and a symbolic value on objects of these classes yielding an internal representation of the (unevaluated) sum.

This representation solves the aforementioned problems custom virtual machines and instrumentation face: The program is not executed in a nested virtual machine. Thus, it can be executed comparatively fast. In contrast to a concolic execution, we do not need to map the concrete state of a running program to a symbolic state and hence eliminate the corresponding indirection.

First, this is beneficial for programmers developing new features for Mulib: They can test and

debug new functionalities by directly writing code using Mulib-library classes. Then, if they are content with the implementation of library classes, they can write a program transformation. In this program transformation the original program is transformed to a program in which the newly developed features are used. Thus, the technical overhead of dealing with and debugging byte-code is mitigated. Moreover, the overall backend can be very flexible, as will be demonstrated in the following. We allow for not only concolic execution, but also performing a pure symbolic execution using incremental constraint solvers in an efficient manner. Hence, contrary to concolic execution, we are not necessarily bound to concrete values during symbolic execution and avoid the need for relabeling.

3 Overview of Mulib

In Muli, symbolic execution is restricted to code in a *search region*. It is desired that operations outside of this search region operate with the usual performance of the JVM. This is challenging since within the search region the runtime must account for complex behavior such as the enforcement of constraints as well as backtracking operations [17]. Mulib approaches this issue by providing a separate search region program that is evaluated instead. An overview of Mulib is provided in the following. First, in Subsection 3.1 it is illustrated how Mulib can be used as a runtime for Muli. Then, in Subsection 3.2, a high-level description of how to formulate search regions in Mulib is provided. It is explained how solutions can be extracted from such a search region in Subsection 3.3. Thereafter, the overall workflow for extracting solutions from a search region is described in Subsection 3.4, the concrete elements of which are detailed in the subsequent sections. Subsection 3.5 then summarizes the architecture while in Subsection 3.6 the differences between Muli$_{sjvm}$ and Mulib are discussed.

3.1 Using Mulib as an Implementation of Muli

Muli initially was implemented via Muli$_{sjvm}$ using a custom compiler and a custom SJVM. Mulib strives to make custom tooling optional and relies on usual Java bytecode. Figure 1 illustrates how Mulib can be used as an implementation of Muli.

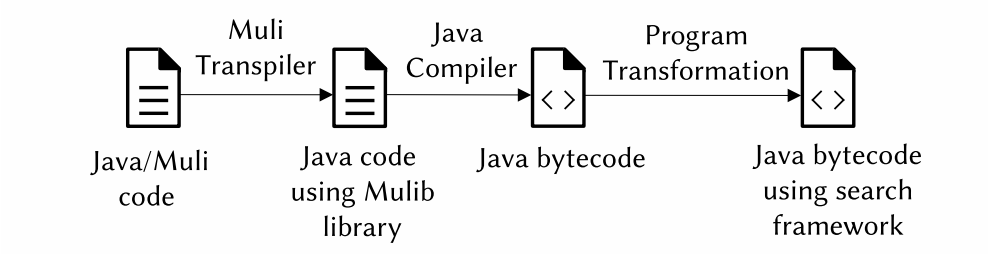

Figure 1: The use of a transpiler for using Muli with Mulib.

Developers can code in Muli and make use of its extended syntax to allow for, e.g., the `free` keyword and the easy creation of search regions using lambdas [18]. This code can then be transpiled to make use of the more low-level Mulib library. Developers that do not wish to make use of the transpiler or Muli syntax can also directly use the Mulib library to formulate and call search regions (see Subsection 3.2). In both cases, the Java source code using Mulib as a library is compiled to usual Java bytecode. Then, a special program transformation is applied that takes the Java bytecode with references to the Mulib framework as input and outputs additional classfiles in Java bytecode. These additional classfiles account for all features of symbolic execution, such as constraints, nondeterminism, and symbolic values.

The following subsections abstract from the transpiler and instead demonstrate the direct use of Mulib, i.e., writing pure Java code using Mulib as a library.

3.2 Search Regions in Mulib

For users of Mulib, it is sufficient to use the functionality provided by the `Mulib` class to initialize symbolic values, assume that constraints hold, or stop the evaluation of certain program branches. In Mulib, a search region is a static method that can contain multiple parameters. Consider the following dummy search region, in which many of Mulib's (and thus Muli's) features are represented:

```
1  public static int dummyMethod() {
2     int i = Mulib.freeInt(1,9);
3     A a = Mulib.freeObject(A.class);
4     A[] as = Mulib.freeObject(A[].class);
5     // Alternative: rememberedFreeObject("as", A[].class);
6     Mulib.remember(as, "as");
7     Mulib.assume(i == as.length);
8     A a = as[i-1];
9     as[i-1] = null;
10    int otherInteger = Mulib.freeInt();
11    if (otherInteger == i) throw Mulib.fail();
12    return i;
13 }
```

Listing 1: A method demonstrating how to create symbolic values and add constraints in Mulib.

In line 2 we initialize a free int value. We use a static *indicator method*, `Mulib.freeInt(int, int)` that also sets the variable's lower bound to `1` and its upper bound to `9`. It is shown how we can initialize an object of type `A` for which all fields have symbolic values and which is lazily initialized [29] in line 3. In turn, in line 4 we initialize a free array of type `A[]` [51, 53]. Line 6 creates a snapshot of the array of elements `as`, i.e., the array at this current state will be remembered under the name `"as"`. In line 7, the constraint that the free int `i` is equal to the length of the array is pushed onto the constraint stack. This means that in the subsequent execution, `i` must be equal to `as.length`, otherwise this path of the execution is invalid and will be discarded. Thereafter, we get the last element from the array `as` (line 8) and set the respective position in as to `null` (line 9). Note that this change will not be reflected in the snapshot of `as` that is called `"as"` that was created in line 6. In line 10 we create a new free int and compare it to `i` spawning a choice point with two choice options in line 11. A choice point is created because a symbolic value is used in the condition-part of an `if` statement. Either the first int `i` is equal to the second one, as is represented by one choice option, or not, as represented by the second choice option. If the values are equal, the choice option is discarded using `Mulib.fail()`. When throwing this special exception, the current evaluated choice option is declared to be invalid and the next choice option is retrieved. Otherwise, `i` is returned (line 12).

3.3 Executing Search Regions and Extracting Solutions in Mulib

Since Java itself is not capable of symbolic execution, the search region must be called in a special way so that the Mulib runtime performs symbolic execution. In fact, most indicator methods, called from Java code outside of a search region, will simply throw an error. This design decision has been made so that attempts to initialize symbolic values do not leak into normal Java code, leading to unexpected errors. Instead, we must make use of another set of methods from the `Mulib` class, an excerpt of which is given in the following:

```
public final class Mulib {
   public static List<PathSolution> getPathSolutions(
         Class<?> methodOwnerClass,
```

```
        String methodName,
        MulibConfigBuilder mb,
        Class[] argTypes,
        Object[] args) { ... }
    public static String generateTestCases(
        Class<?> methodOwnerClass,
        String methodName,
        MulibConfigBuilder mb,
        Class[] argTypes,
        Object[] args,
        Method methodUnderTest,
        TcgConfigBuilder tcgb) { ... }
    public static List<Solution> getUpToNSolutions(
        Class<?> methodOwnerClass,
        String methodName,
        MulibConfigBuilder mb,
        int N,
        Class[] argTypes,
        Object[] args) { ... }
     public static Stream<Solution> getSolutionStream(
         Class<?> methodOwnerClass,
         String methodName,
         MulibConfigBuilder mb,
         Class[] argTypes,
         Object[] args) { ... }
     public static SolutionIterator getSolutionIterator(
         Class<?> methodOwnerClass,
         String methodName,
         MulibConfigBuilder mb,
         Class[] argTypes,
         Object[] args) { ... }
     public static MulibContext getMulibContext(
         Class<?> methodOwnerClass,
         String methodName,
         MulibConfigBuilder mb,
         Class<?>... argTypes) { ... }
    ...

}
```

All methods require that the user specifies the class containing the search region as well as the method name of the search region. The search region should be a static method and optionally can have multiple parameters. A `MulibConfigBuilder` is passed to provide a configuration of how Mulib should behave.[4]

`Mulib.getPathSolutions(...)` returns a list of `PathSolution`s. The content of the path solution is depicted in Figure 2. A path solution contains data for an execution path through the search region for which the constraint stack is satisfiable. It might also be a `ThrowablePathSolution`, which indicates that the search region has been left by throwing a `Throwable` [28]. Each `PathSolution` comes with a `Solution` object carrying the return value of the search region. Moreover, each `Solution` object comes with a `Labels` object from which remembered variables can be retrieved. For instance, Listing 1 stores an array using the name `"as"`. This can be retrieved from the `Labels` object using `Labels.getLabelForId("as")`. Extracting `PathSolution`s is sufficient for checking the assertions in a search region. In fact, `Mulib.generateTestCases(...)` reuses the method extracting path solutions to generate a String representation of executable test cases. However, if the aim of the search region is not checking the assertions of the code at hand or generating test cases, path solutions are not always adequate. Consider, for instance, this search region to solve the

[4] See Subsection 8.2 for an overview of the configuration options.

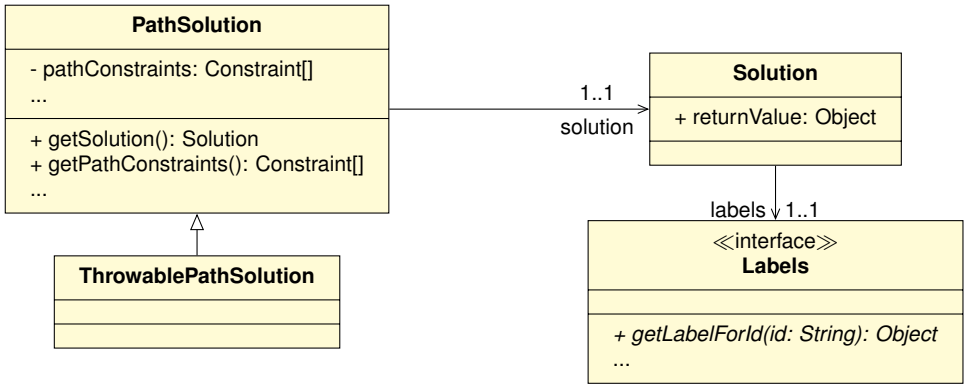

Figure 2: The structure of `PathSolution` and `Solution`.

well-known 8-Queens problem [22]:

```
1  public static Queen[] solve() {
2     final int n = 8;
3     Board board = new Board(n);
4     Queen[] qs = new Queen[n];
5     for (int i = 0; i < n; i++) {
6        Queen q = new Queen();
7        q.x = i;
8        q.y = Mulib.freeInt();
9        qs[i] = q;
10    }
11    for (int i = 0; i < n; i++) {
12       Mulib.assume(board.isOnBoard(qs[i]));
13       for (int j = i+1; j < n; j++) {
14          Mulib.assume(!board.threatens(qs[i], qs[j]));
15       }
16    }
17    return qs;
18 }
```

In this program, we create a board with the dimensions 8×8 and 8 queens (lines 2–4). Thereafter, we initialize the coordinates for all queens, where the x-coordinate is fixed and the y-coordinate is free (lines 5–10). Free values are initialized using indicator methods, such as `Mulib.freeInt()`. Thereafter, we assure that each queen has a valid y-coordinate (line 12) and that each queen does not threaten another queen (lines 13–15).[5] Noteably, there only is one valid path through the entire program. This is because there is no case distinction involving a symbolic value which would cause nondeterminism. Muli and Mulib will thus return exactly one path solution for this problem. However there are 92 distinct solutions for the 8-queens problem. In other words: The labeling given for the path solution is not the only valid labeling of this path. In consequence, there are multiple possible solutions. In Mulib, we can extract all 92 solutions by using `Mulib.getUpToNSolutions(...)` as shown here:

```
1 MulibConfigBuilder mb =
2    MulibConfig.builder()
3       .setSOLVER_GLOBAL_TYPE(Z3_INCREMENTAL)
4       .setSEARCH_MAIN_STRATEGY(DFS)
5       .setSEARCH_ADDITIONAL_PARALLEL_STRATEGIES(BFS);
6 List<Solution> result =
7    Mulib.getUpToNSolutions(
```

[5] Here, we abstract from the implementation of the methods of `Board`.

```
14

8       NQueens.class, "solve", mb, 93);
```

We first create a `MulibConfigBuilder` for which we specify how the search region should be explored (lines 1–5). In the given example, we use Z3 [24] as an incremental constraint solver (line 3), choose Depth-First Search (DFS) as a search strategy (line 4) and additionally start a second thread in which we additionally evaluate the search region using Breadth-First Search (BFS) (line 5).[6] Even though we specify that we want 93 solutions, only 92 will be returned for this problem.

This approach is generalized by providing the methods `Mulib.getSolutionStream(...)` and `Mulib.getSolutionIterator(...)`. Both methods can be used to incrementally explore the search region, i.e., reinvoke the search region method. In consequence, an undetermined number of solutions can be extracted in an on-demand fashion (see Dageförde and Kuchen [20] for an analogous explanation for Muli$_{sjvm}$). In other words: The user can perform stream operations defined on `java.util.Stream` on the `Solution` objects or iterate solutions using `java.util.Iterator`. In both cases `Solution` objects are calculated from the search region lazily. The respective algorithm to extract multiple solutions from a single path, accounting for the continuation of the search upon user request, is explained in Subsection 7.3.

3.4 Overall Workflow

This subsection will describe the overall workflow when executing Mulib. Details to the various components are given in the subsequent sections. The overall workflow can be divided into two stages: First, a preparatory stage in which the search region is set up to allow for symbolic execution. Second, the symbolic execution itself.

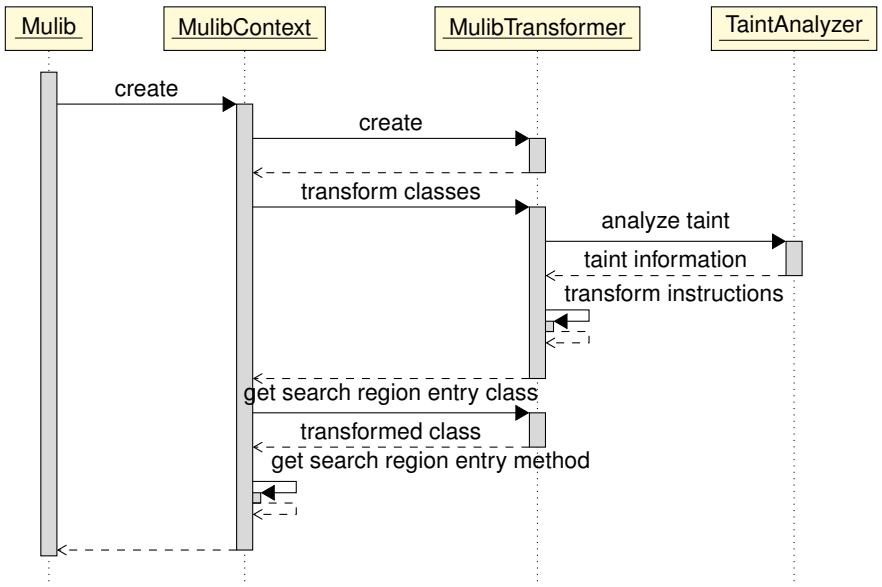

Figure 3: Abstract view on the workflow for setting up an instance of `MulibContext`.

The preparatory workflow is depicted in Figure 3. When calling, e.g., `Mulib.getPathSolutions(...)` a temporary instance of `MulibContext` is created. A `MulibContext` holds a transformed version of the method that shall be symbolically executed and can be used to extract `PathSolution`s or

[6] An overview of configuration options is given in Subsection 8.2. `MulibConfig.builder()` returns a default configuration.

Solutions by starting the workflow depicted in Figure 4. This MulibContext creates an instance of MulibTransformer to perform a program transformation on the class carrying the method that is the access point to the search region (in the following denoted *search region method*) and the (implicitly) used classes. The transformed classes are used in the search region but not outside of the search region. In other words: the transformed classes are the *search region representation*s of the original classes, also called *partner classes* [52]. More details on the program transformation are given in Section 5. For now it suffices to say that the search region representation allows us to (1) represent symbolic values where Java would only allow concrete values and (2) account for the creation of choice points and the decision on which choice options are satisfiable and shall be explored next. To achieve this, an intra-procedural taint analysis is used to determine which instructions and values potentially are symbolic.

We retrieve the partner class containing the search region method from the MulibTransformer that transformed the search region. Using reflection, we then extract the search region method from the transformed class. After executing these steps, the MulibContext is now completely set up and accepts different arguments for invoking the search region (if the search region has any parameters).

If the MulibContext was not created using Mulib.getMulibContext(...), it was created using a method such as Mulib.getPathSolutions(...). In this case, MulibContext.getPathSolution(...) is called next, starting the second stage of the workflow. This is depicted in Figure 4. In a first step, the arguments are transformed to their search region representation. The transformed program then contains types (see Section 4) and operations for which the Mulib search framework can conduct symbolic execution. Furthermore, the MulibTransformer retrieves the static variables of the transformed classes. They will be wrapped in an instance of StaticVariables so that they can be accessed and changed by multiple executors simultaneously (see Subsection 5.7 for more details).

Thereafter, a set of factories, namely ValueFactory, CalculationFactory, and ChoicePointFactory are created. Implementations of these factories define the way values are generated, calculations are performed, and choice points are created. For instance, SymbolicValueFactory generates purely symbolic values while ConcolicValueFactory additionally generates a label for the symbolic values. More details on concolic execution in Mulib are given in Subsection 8.1. More details on the ValueFactory and CalculationFactory are given in Subsection 6.5 while the ChoicePointFactory is described in Subsection 6.4. Then, a MulibExecutorManager is created which is responsible for managing MulibExecutors. A search tree and a choice option deque are initialized to hold unevaluated choice options [52]. MulibExecutors, in turn, extract PathSolutions from the search region in a loop and represent search strategies. They first copy the transformed arguments that are passed to it and thereafter invoke the method handle holding the search region with these arguments. Thereafter, a SolverManager (not depicted) is used to label the solution, i.e., assign concrete labels to the search region values. The loop terminates if there are no unvisited choice options in the search tree/deque or if a global budget has been exceeded (see Subsubsection 8.2.3). The returned path solutions then are the result to the initial call of Mulib.getPathSolutions(args).

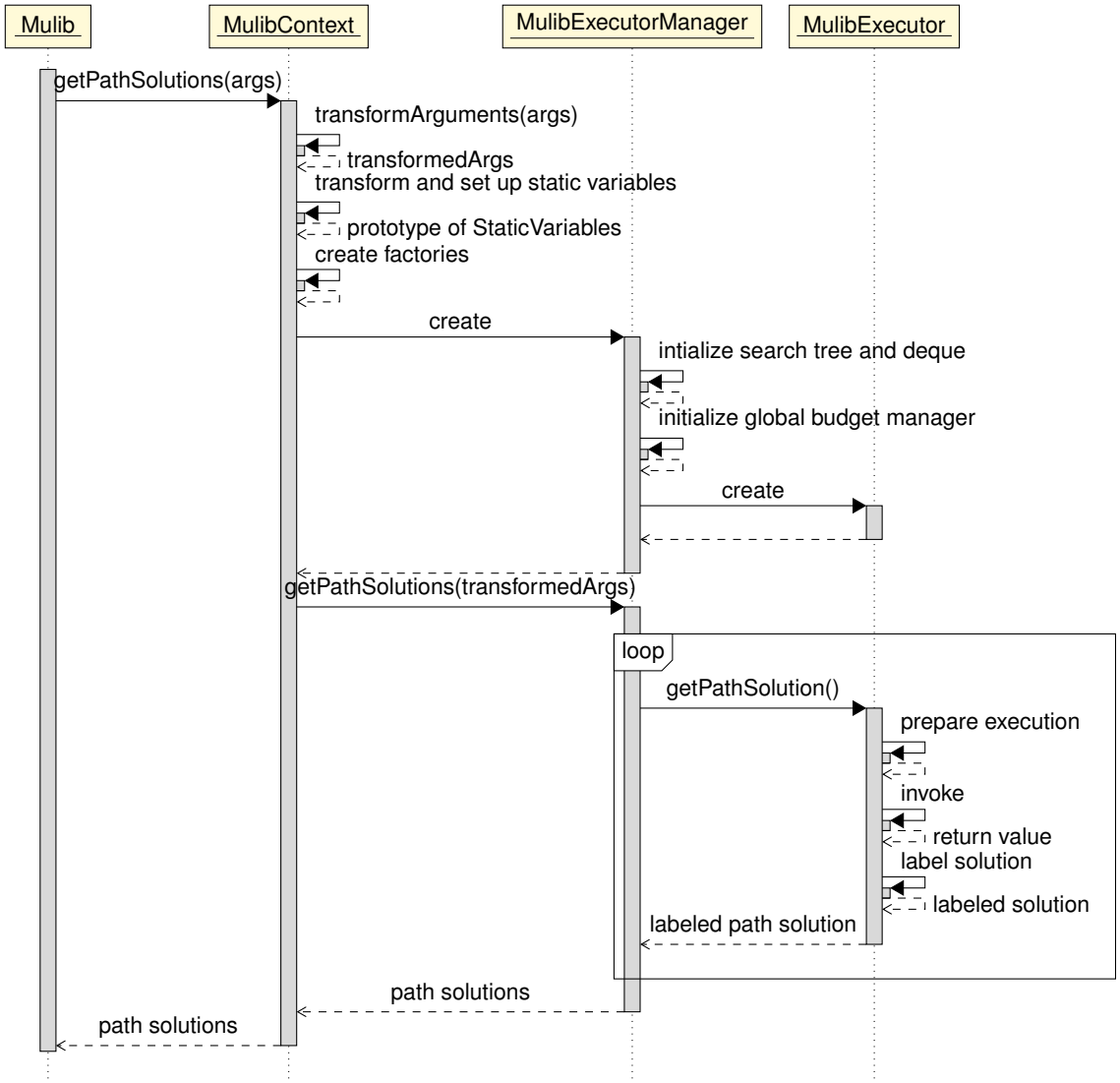

Figure 4: Abstract view on the workflow for deriving path solutions using one `MulibExecutor`.

3.5 Architecture of Mulib

Overall, the architecture of Mulib is summarized in Figure 5. The access point for any program using Mulib is the `Mulib` class. A configuration can be specified and a context can be spawned. In the context, a program transformation is conducted that reads the (byte-)code of the classes that are used in the search region and transforms them (see Section 5) to use types (see Section 4) specific to the Mulib search framework. The result (byte-)code can then be used to start the search framework using the search region method. The search framework (see Section 6) consists of an executor manager that holds references to the search tree, a deque of choice options, and aforementioned factories. The executor manager furthermore spawns one or many executors, each of which represents a search strategy. Each executor has access to the aforementioned factories (see Subsections 6.5 and 6.4), the search tree (see Subsection 6.1), the choice option deque (see Subsection 6.6) and contains its own instance of an adapter used to connect to an external constraint solver (see Section 7). Finally, each executor also has its own container for static variables.

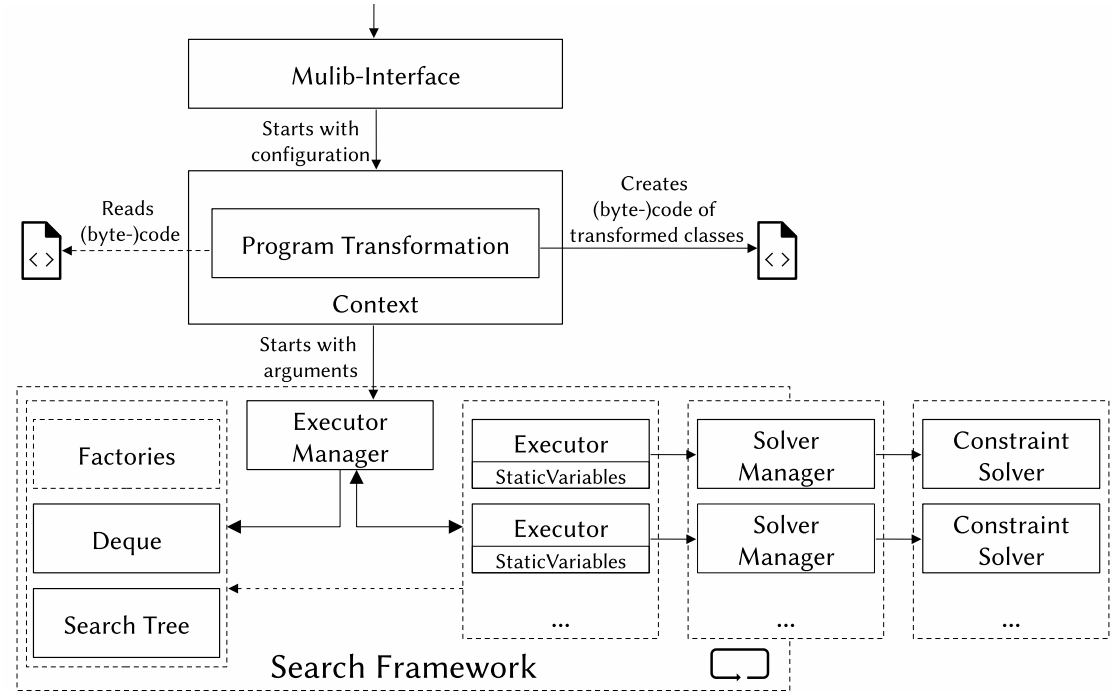

Figure 5: A conceptual view on the core components of Mulib.

3.6 Differences to Muli$_{sjvm}$

Muli$_{sjvm}$ relies on a custom compiler which generates bytecode that can be executed by a custom SJVM. This approach has several downsides: First, the custom compiler and the SJVM are additional tooling. Developers and organizations strive not to depend on too many technologies. In consequence, the adoption of CLOOP is hindered if it depends on such custom tooling. Adding to this point, Muli, as a superset of Java 8, also needs custom IDE support. The `free` keyword, although convenient, causes a syntax error for Java IDEs. To still utilize an IDE with Muli, customizations must be applied. Moreover, the performance of the custom SJVM is lacking both in terms of memory consumption and speed [52].

In contrast, Mulib decouples the CLOOP-specific language from the execution engine. When paired with a transpiler transforming Muli code to the approach using indicator methods (recall Figure 1), Mulib is an implementation of Muli. However, Mulib does not need to rely on the additional `free`-keyword. This keyword is represented by indicator methods that do not trigger any errors in Java IDEs. Muli$_{sjvm}$ and Mulib both offer several methods for exploring the search region. In Muli, a parameterless lambda-expression is specified that shall be symbolically executed. In Mulib, the user must specify the class declaring the search region method, the name, as well as the arguments to the search region method. Admittedly, this is more error-prone. This issue can be dealt with by creating the aforementioned transpiler from Muli code to Java code using Mulib. The reason why the `Mulib`-API is defined in the current way is that, to the best of our knowledge, there is no future-proof way of retrieving the (byte-)code of a lambda stored in an object at runtime. However, the respective transpiler is relatively straight-forward: One must replace the `free` keyword with appropriate indicator methods, such as `freeInt();`.

With Mulib, new language structures can be experimented with in a simplified way. Solely a new indicator method must be created. In contrast, in Muli$_{sjvm}$, the grammar of Muli as well as the custom compiler must be adapted. For instance, the straight-forward definition of an upper and

lower bound as well as the `assume(boolean)`-method were added. In Muli$_{sjvm}$, one had to choose the more verbose structure of `if (!constraint)throw Muli.fail()`. This construct will create a choice with two choice options: Either the constraint holds or not. The branch where the constraint does not hold is immediately marked as unsatisfiable using `Muli.fail()`. While this option still is possible in Mulib, simply assuming a constraint holds is more intuitive in many scenarios.[7]

Moreover, as a conceptual extension, Mulib introduces the differentiation between `PathSolution`s and `Solution`s. Muli$_{sjvm}$'s `Solution` object now is called `PathSolution`. A `PathSolution` is relevant for checking `assert`ions and generating test cases. There might be multiple other valid labels on a `PathSolution` but for the aforementioned use cases any valid solution to a path through the search region usually is sufficient. For the use case of solving constraint problems, retrieving `Solution`s is a more suitable approach, since, generally, the path through the program is not the primary concern, but rather, valid labels are interesting.

Finally, Muli$_{sjvm}$ allowed for defining variables and even object and class fields to be potentially `free`. This can lead to symbolic values leaking into the Java code outside of the search region. Since the deterministic runtime cannot work with symbolic values, a crash is imminent. This does not happen when initializing the free value, but rather, when operating on it outside of a search region. In contrast, Mulib enforces that symbolic values are solely created and used within a search region. This is done by introducing a separate search region representation that can initialize symbolic values. Trying to initialize symbolic values outside of a search region will throw an immediate and clear error. Furthermore, the labeled values and objects will always have usual Java types which means that outside of a search region, all code will work exactly as it would without using Muli(b). The benefit of having a separate representation for classes in the search region becomes especially apparent when regarding static variables. A further discussion on this can be found in Subsection 5.7.

[7] However, this is currently still done creating a choice point (see Subsubsection 4.1.2 for an explanation).

4 Search Region Representations

In Muli, values can be symbolic, i.e., the concrete value is not known during execution. However, the primitive types of Java [33], such as `int`, `double`, and `boolean`, can only carry concrete values. This also extends to, e.g., free arrays in Muli, having a free length [51], which is not representable using Java arrays. In consequence, to symbolically execute Java code, a new representation is desirable that can differentiate between symbolic and concrete values. Operations on concrete values should be performed so that a concrete value is the output while operations involving symbolic values should create an appropriate symbolic description of the operation. Furthermore, outside of said search regions, normal Java types, such as primitives and arrays, should again be used to avoid causing any overhead.

Figure 6 gives an overview on the most important library types. Subclasses of `Sarray` and `PartnerClassObject` are generated for individual search regions. As shown in Subsection 3.4, Mulib first transforms the code used in a search region to code using special types. These types are able to express symbolic values and have symbolic operations attached to them, such as the addition of two, potentially symbolic, values. This section explains the types in more detail. In contrast, Section 5 describes how we inject the respective types and calculations into the search region. First, in Subsection 4.1, the representation of primitives is explained. Thereafter, the approach to represent objects is summarized in Subsection 4.2. Last, the differences between Muli$_{sjvm}$ and Mulib are explained in Subsection 4.3.

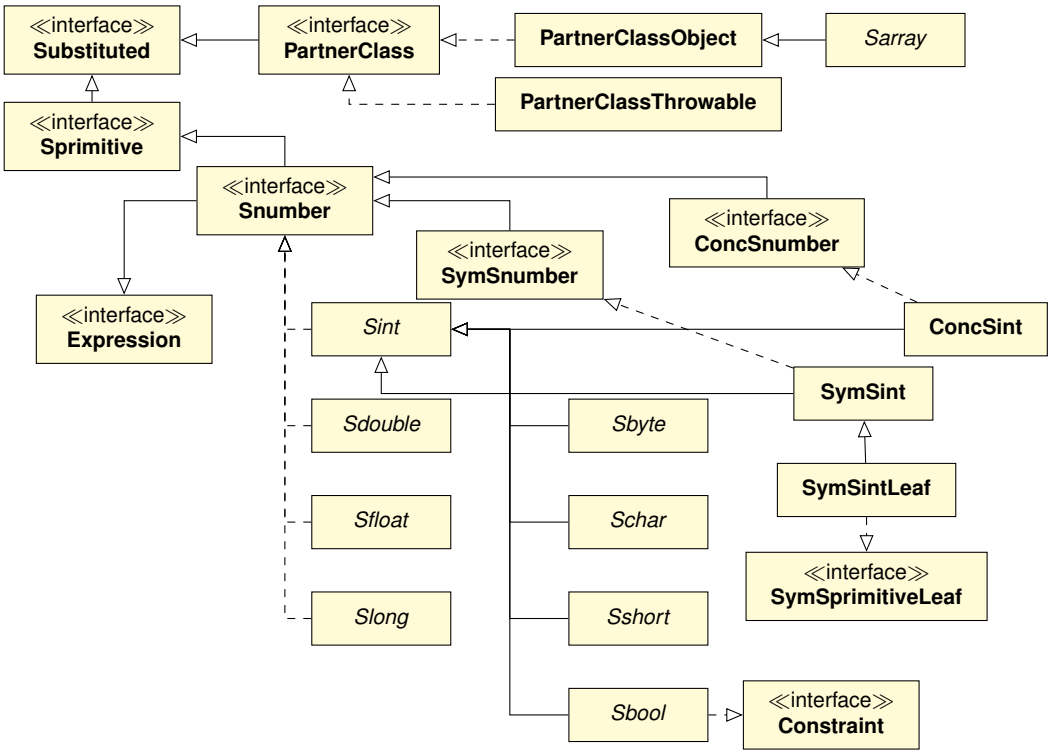

Figure 6: An excerpt of the most important types in Mulib.

4.1 Primitives

For each of the primitive types in Mulib, there is a special class in Mulib. For instance, the counterpart of `int` is `Sint`.[8] `Sint` is the abstract superclass of `ConcSint` and `SymSint`. `ConcSint` represents concrete `int` values in the search region while `SymSint` wraps an `Expression` in the form of a DAG. The other `Snumber`s, such as `Sdouble`, are not dedicatedly illustrated in Figure 6, but the subtypes are structured analogously to `Sint`.

On the other hand, `Sbool` represents the special case of boolean expressions in the search region. `ConcSbool` is either `ConcSbool.TRUE` or `ConcSbool.FALSE`, while `SymSbool` wraps a DAG of type `Constraint`.

Note that multiple types, such as `Schar` and `Sbool` are also subclasses of `Sint`. The reason for this is that in Java, bytecode instructions do not differentiate them for the most part. For instance, two `char`s can be added in Java via the $iadd$ bytecode instruction. In theory, also `boolean`s can be added to other `int` values via $iadd$, even though the Java syntax disallows this.

4.1.1 Expressions

The leaves of aforementioned DAGs are either `ConcSnumber`s or `SymSprimitiveLeaf`s that are `Snumber`s. An overview of possible non-leaf nodes in this tree are given in Table 1. The table displays the class name representing the respective non-leaf node of the DAG as well as a brief description of it. A bytecode instruction also is given. These expressions are created by Mulib if a symbolic value would become part of an execution of a respective bytecode instruction [52]. Consider, for instance, the execution of the following bytecode snippet:

```
iload 0
iload 1
imul
```

Two integer values are loaded from variables via the $iload$ instruction. Thereafter, they are multiplied. Mulib transforms the program (see Section 5) in a way that, if at least one of the variables contains a symbolic value, the expression `Mul` is created instead. This expression then is wrapped by an instance of `SymSint`.

All of the expressions can be mapped to a Java bytecode instruction that is transformed to be represented by a method call (see Section 5). This method call then can return wrapped `Expression`s if the argument list contains any symbolic values.

4.1.2 Constraints

The leaves of the DAG representing constraints are either `ConcSbool` or `SymSboolLeaf`s. In symbolic execution, constraints are generated for choice options. Choice options are mostly created when we make conditional jumps, e.g., via `if` statements. In Java, there are several bytecode instructions for such conditional jumps. Table 2 covers the various non-leaf nodes for constraints in Mulib, gives a brief description of the respective constraints, and, if possible, enumerates the bytecode instructions creating said constraint.

However, note that Java does not have any immediate bytecode instructions for, e.g., con- or

[8] `Sint` has been chosen as a name as the `int`-type has been substituted for as long as the search region is explored, as such it is a **s**ubstitute for the `int` type.

Name	Description	Bytecode Instruction
Div	Division of two expressions ($/$)	$ddiv, fdiv, idiv, ldiv$
LogicalShiftRight	Shift a number to the right logically ($>>>$)	$iushr, lushr$
Mod	Calculating the modulo ($\%$)	$drem, frem, irem, lrem$
Mul	Multiplication of two expressions ($*$)	$dmul, fmul, imul, lmul$
BitwiseAnd	Bitwise "and" operation ($\&$)	$iand, land$
BitwiseOr	Bitwise "or" operation ($\|$)	ior, lor
BitwiseXor	Bitwise "xor" operation ($\hat{}$)	$ixor, lxor$
ShiftLeft	Shift a number to the left arithmetically ($<<$)	$ishl, lshl$
ShiftRight	Shift a number to the right arithmetically ($>>$)	$ishr, lshr$
Sub	Subtraction of two expressions ($-$)	$dsub, fsub, isub, lsub$
Sum	Addition of two expressions ($+$)	$dadd, fadd, iadd, ladd$
Neg	Negation of an expression ($-$)	$dneg, fneg, ineg, lneg$

Table 1: The non-leaf expressions in Mulib.

Name	Description	Bytecode Instruction
And	"and" operation ($\&\&$)	–
BoolIte	Represents one of two values based on a constraint the label of which is not yet known	–
Eq	"equality" operation ($==$)	$ifeq, if\_icmpeq, if\_acmpeq,$ $tableswitch, lookupswitch$
Equivalence	"equality" operation with two booleans ($==$)	if_icmpeq
Implication	"implication" operation	–
In	"element of" operation	–
Lt	"less-than" operation ($<$)	$iflt, ifgt, if\_icmplt, if\_icmpgt$
Lte	"less-than-equals" operation ($<=$)	$ifle, ifge, if\_icmple, if\_icmpge$
Not	Negates a constraint ($!constraint$)	$ifne, if\_icmpne, if\_acmpne$
Or	"or" operation ($\|\|$)	–
Xor	"xor" operation ($\hat{}$)	–

Table 2: The non-leaf constraints in Mulib. Expands Dageförde and Kuchen [18].

disjoining `boolean`s. Instead, Java will create several jumps to initialize new booleans as is shown in the following example:

```
boolean and(boolean b0, boolean
    b1) {
    return b0 && b1;
}
```

```
boolean or(boolean b0, boolean
    b1) {
    return b0 || b1;
}
```

When compiling these two methods, the following (simplified) bytecode is generated:

```
1     L0
2       ILOAD 0
3       IFEQ L1
4       ILOAD 1
5       IFEQ L1
6       ICONST_1
7       GOTO L2
8     L1
9       ICONST_0
10    L2
11      IRETURN
```

```
1     L0
2       ILOAD 0
3       IFNE L1
4       ILOAD 1
5       IFEQ L2
6     L1
7       ICONST_1
8       GOTO L3
9     L2
10      ICONST_0
11    L3
12      IRETURN
```

In both cases, rather than executing some specific Java bytecode instructions for calculating `b0 || b1` or `b0 && b1` (which do not exist), conditional jumps are performed pushing either a `0` or a `1`, signaling that the result is either `false` or `true`. For instance, in the case of `and(boolean, boolean)`, the first `boolean` is loaded (line 2), it is checked if it equals to `0` and thus `false` (line 3). If this is the case, we jump to L1 (line 8) to push `0` and return this result (line 11). Only if the `boolean` is not `0`, we continue with checking whether the second boolean is `0` (lines 4 and 5). If this is not the case, we push `1` (line 6) and return (lines 7 and 11).

Hence, constraints such as `And` or `Or` are currently solely used internally in Mulib. Future research might work on *folding* such conditional jumps so that we instead of creating a choice point, a new symbolic value of type `SymSbool` is created. This can potentially speed up the execution vastly (see Subsection 9.4).

4.2 Search Region Representation for Objects

While the built-in primitive types can be dedicatedly modeled via special classes, user-defined classes and arrays must also be considered. For both, information from the search region is required.

4.2.1 User-Defined Classes and Interfaces

In Java, all objects have the common supertype `Object`, even without explicitly declaring it [33]. `Object` defines some crucial methods that are used throughout Java. The Java compiler even allows calling `Object`-methods for interface types, that, by definition, usually cannot extend a class [28].

Similarly, the search region representation of classes has to account for complex behavior, such as symbolically representing the object for/in a constraint solver [53]. Another example is that it is desirable to have custom methods, e.g., for transforming a class from outside the search region into its partner class and copying a `PartnerClass`. Those steps are executed if the search region has any parameters (see Figure 4). It thus is convenient to have a set of methods that can be invoked on all partner class objects.

The search region representation of user-defined classes, aside from classes extending `java.lang.Throwable`, all have the common superclass `PartnerClassObject`. To assure that methods for `PartnerClassObject` can also be called for interfaces, an interface `PartnerClass` was added. User-defined interfaces will not extend `PartnerClassObject` but `PartnerClass`. `PartnerClass` thus is the search region-equivalent of `Object`. In turn, the role of `PartnerClassObject` is to implement the various methods of `PartnerClass` that are, e.g., related to the state of an object, e.g.,

whether or not the object is currently represented for/in the constraint solver [53] or whether it is to-be lazily initialized [29]. `PartnerClassThrowable` is a special case for those classes that extend `Throwable`. In Java, instances of `Throwable` represent some `Error` or `Exception` and can be *thrown* as an alternative to leaving a method using a return statement [33]. Instances of `Throwable` also can be *caught*, if they have been thrown. However, in Java `Throwable` is a class. Since the JVM does not allow for multiple inheritance [33], partner classes of subclasses of `Throwable` must not extend `PartnerClassObject`. Instead, `PartnerClassThrowable` is extended, which, in turn, extends `Throwable`. Thus, `PartnerClassThrowable` is implemented like `PartnerClassObject`, with the exception that former class extends `Throwable`.

In Mulib, partner classes are generated using the transformation indicator `__mulib__`. For instance, the partner class of a class `A` with the superclass `Object`, if used in a search region, will receive the name `__mulib__A` and the superclass of this partner class will be `PartnerClassObject`. Instead of `A`, in the search region, `__mulib__A` is used. In consequence, if a search region method has a parameter of type `A`, it will be transformed into a type `__mulib__A` before invoking the search region (see Figure 3). There are several things to consider when transforming the bytecode of such a class `A` to its search region counterpart. These considerations are further explained in Section 5.

On the other hand, arrays are transformed to subclasses of the special class `Sarray`. The issue with Java's arrays is that for accessing them, only concrete integer numbers can be used. This is because in Java there is no such concept as a symbolic value. Furthermore, it must be possible to create arrays with a symbolic length. The `Sarray` class is implemented in a way allowing for both symbolic accesses as well as a symbolic length. For the primitive array types, such as `int[]` and `boolean[]`, respective subclasses have already been provided in the form of `SintSarray` and `SboolSarray` (not depicted). On the other hand, for arrays with reference-typed elements, such as `A[]` or `A[][]`, subclasses of the types `PartnerClassSarray` and `SarraySarray`, are created respectively. In consequence, if, in the search region, arrays of the aforementioned types are used, the `MulibTransformer` (see Figure 3) also synthesizes the classes `__mulib__A__mulib__PartnerClassSarray` and `__mulib__A__mulib__2__mulib__SarraySarray`. Note that, again, the transformation indicator string is used in a manner to assure that ambiguous names are avoided. Inputs to the search region of array types consequentially also are transformed to synthesized (or predefined) `Sarray`s.

4.2.2 Partner Class Object Constraints

`PartnerClassObjectConstraint`s are constraints that restrict data of an object, such as the elements of an array or the values of an object's fields. Figure 7 gives an overview of the implemented constraints of this type. `PartnerClassObjectConstraint` is not a subtype of `Constraint` since it cannot be arbitrarily combined with other constraints. Such a combination, e.g., a disjunction of a `Constraint` and a `PartnerClassObjectConstraint`, would be meaningless, as is explained in the following. `PartnerClassObjectConstraint`s are created when dealing with selecting from or storing in an array using a symbolic index or in the case of *symbolic aliasing*. Symbolic aliasing denotes the concept that, in Mulib, it can occur that two objects have the same identity. Objects are endowed with an identifier represented by a number. This identifier can also be a symbolic value and it can potentially equal one of the existing identifiers. If this is the case, these two objects are equal in identity. Consequently, if, e.g., the fields of two objects are accessed which have the same identifier, it must be assured that the returned values equal. Additionally, if one of these objects is mutated by assigning a new value to its field, this side-effect must also be propagated to the other object. In Mulib, this is done using constraint which are subtypes of `PartnerClassObjectConstraint`. The interested reader is referred to Winkelmann and Kuchen [53] for more precise explanations of the underlying concepts.

A `PartnerClassObjectInitializationConstraint` is posed when representing a non-array object for the constraint solver. An object can be represented for the solver for various reasons. For

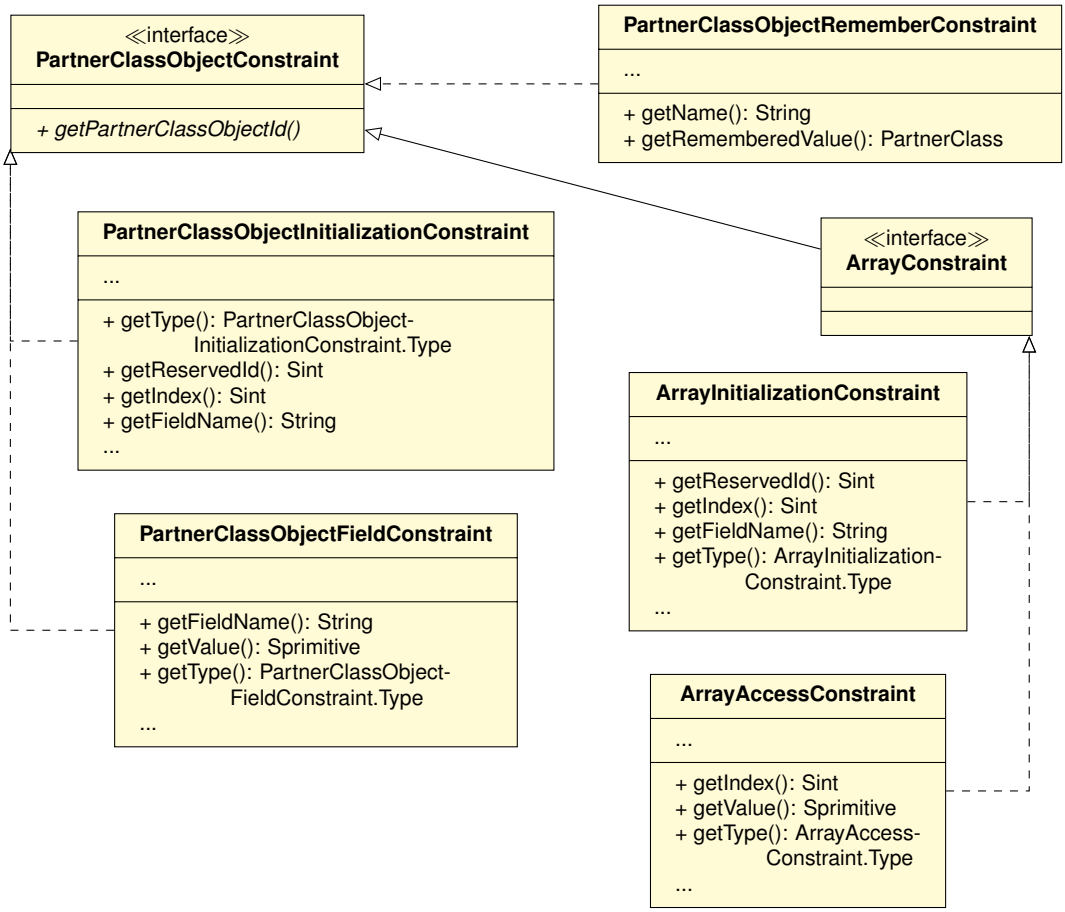

Figure 7: The constraints that restrict the metadata of partner classes.

instance, if it is contained in an array of objects and this array is selected from using a symbolic index, the content of this array must be represented for the solver. This allows for delegating the decision on the identity of the retrieved object to the constraint solver [53]. If an array of objects is represented for the constraint solver due to an access with a symbolic index, the result of invoking `getType()` for the elements of the array that is being represented would be `SIMPLE_PARTNER_CLASS_OBJECT`. If now, using a symbolic index, an element is selected from the array of objects, a new object is created. The returned object of this array also is represented for the constraint solver. It's type would be `PARTNER_CLASS_OBJECT_IN_SARRAY`. This is because it was created to refer to one of the elements of the array. The `SolverManager` will then restrict its symbolic identifier to equal one of the identifiers that is already represented in the array. In other words: The newly created object is a *symbolic alias* of the objects in the array which, in turn, are called *aliasing targets* [53]. The solver backend ensures that, if two objects have the same identifer, their fields' contents must be the same [53]. In the given instance, `getIndex()` returns the index for which the new object is created in the array. Analogously, `PARTNER_CLASS_OBJECT_IN_PARTNER_CLASS_OBJECT` describes the case of `PartnerClassObjectInitializationConstraint` where the object is not taken from an instance of `Sarray`, but from a field of a `PartnerClass` that was represented for the constraint solver. In this case, `getFieldName()` returns the name of the field for which the new object has been created. If the array that is selected from has a concrete length, its content is known. In this case, the selected object is a symbolic alias of one of the elements present in the array when it was initialized. On the other hand, if the array has a symbolic length, it might be that the selected object is a new, previously unknown, object. In this case `getReservedId()` returns a distinct identifier.

Analogously, `ArrayInitializationConstraint` describes the same concept applied to arrays. If an array is represented for the solver, all its elements also are represented for the solver beforehand. Importantly, the fields or array indexes that contain an object as their value are represented using the identifier of this object [53].

After representing a non-array object for the constraint solver, instead of accessing its fields directly, a `PartnerClassObjectFieldConstraint` is pushed to the constraint solver. If a value should be retrieved for a given field, a new symbolic leaf value (see Subsection 4.1) is spawned. This value is included in an instance of the `PartnerClassObjectFieldConstraint` and the type is set to `GETFIELD`. When pushing this constraint to the constraint solver, all aliasing targets are checked for their values in this field. A disjunction is created so that the selected value must equal to one of the values. If instead, a value is stored in a field, the type of `PartnerClassObjectFieldConstraint` is set to `PUTFIELD`. In this case, the result of `getValue()` is the value that is stored in the representation of the partner class object.

Analogously, an array that was represented for the constraint solver will not access its elements. Instead, `ArrayAccessConstraint`s are generated that represent selecting or storing in the array.

Finally, the `PartnerClassObjectRememberConstraint` is a constraint that is sent to the constraint solver for creating a snapshot of an object at a certain stage [45]. For this, the indicator method `Mulib.remember(...)` can be called. The respective object can then be retrieved from a `Solution` object (recall the `Labels` interface in Figure 2). This is useful, for instance, to memorize the original inputs of a method under test (compare with Winkelmann, Troost, and Kuchen [55]). If the object is not subject to symbolic aliasing [53] or lazy initialization [29, 55] a deep-copy of it is created. If the object is subject to symbolic aliasing or lazy initialization, it is represented for the solver using an appropriate initialization constraint. Then, during labeling, only those `PartnerClassObjectFieldConstraint`s and `ArrayAccesConstraint`s are regarded that occur before overwriting the content of a field.

4.3 Differences to Muli$_{sjvm}$

Muli$_{sjvm}$ used the same data structures for both the search region and the deterministic program parts. Yet, since Muli$_{sjvm}$ uses a custom compiler and an SJVM written in Java [17], the representation of individual objects is a rather abstract one: Each and every object is represented by an instance of `Objectref` or `Arrayref`. For example, the former maintains a map of `java.lang.reflect.Field` to `java.lang.Object` representing the field values of the represented object [26]. In contrast, in Mulib, for as long as no symbolic aliasing is involved (see Winkelmann and Kuchen [53]), actual Java objects are the chosen representation.

Furthermore, primitives in Muli$_{sjvm}$ are represented either via wrappers such as `IntConstant`, or, if they are symbolic, `NumericVariable`. This is similar to Mulib, although Mulib also offers specific types for different number types. However, importantly, outside of the search region, regular Java code is executed and primitive data types are used. These are more efficient than using objects for various operations since they can reside on the stack instead of having a reference to the heap [33]. Muli$_{sjvm}$ introduced a separate *concrete* mode, opposed to the *search* mode that is employed in a search region, where non-symbolic values are expected [17]. Since the primitive types of Muli$_{sjvm}$ can be either symbolic or concrete, it is not straight-forwardly possible to have the same performance as outside of the search region, particularly, because free values can also be defined outside of a search region (recall Subsection 3.6).

Muli$_{sjvm}$ already implements some functionality for representing arrays of primitive-typed elements [51]. Mulib extends this support to arrays of reference-typed elements. In consequence, it needs to allow for representing arrays and non-array objects for/in the constraint solver [53].

Finally, Muli$_{sjvm}$ also offered an option to capture the original inputs of a method [55]. However, since Muli$_{sjvm}$ did not offer the possibility of symbolic aliasing, the used approach did not account for it. In contrast, Mulib also offers support for symbolic aliasing during the labeling procedure.

5 Program Transformation

One of Mulib's core components is a program transformer, i.e., an implementation of the interface `MulibTransformer` (recall Figure 5). The task of a `MulibTransformer` is to generate partner classes of the *original* classes that can be loaded using a class loader. In the following, *original class* denotes a class that is specified by the user and that is transformed using the program transformation. Since the Mulib search framework is an own component, the transformer does not generate complex methods by itself and rather just substitutes bytecode instructions with method calls to the Mulib execution framework.

The remainder of this section is organized as follows: First, in Subsection 5.1, an example is provided to illustrate the functionality of the program transformation. Thereafter, in Subsection 5.2, the entry point to the program transformation is described. This is followed by a description of the transformation of classes in Subsection 5.3 and elaborations on how methods are transformed (Subsection 5.4). Subsections 5.5 to 5.7 provide some more details on special cases in the program transformation. Finally, Subsection 5.8 compares the program transformation of Mulib to the compiler and SJVM of Muli$_{sjvm}$.

5.1 An Exemplary Transformation

As an example for a program transformation, consider the following program using Mulib to solve a constraint satisfaction problem:

```
1  public static int[] assign(Machine[] machines, int[] wkloads) {
2      int[] assignments = new int[wkloads.length];
3      for (int i = 0; i < wkloads.length; i++) {
4          int wkload = wkloads[i];
5          int machineIndex = Mulib.freeInt();
6          Machine machine = machines[machineIndex];
7          int capacity = machine.cap;
8          if (capacity < wkloads[i]) {
9              throw Mulib.fail();
10         }
11         machine.load(wkloads[i]);
12         assignments[i] = machineIndex;
13     }
14     return assignments;
15 }
```

More details for this type of assignment problem can be found in Winkelmann and Kuchen [53]. A respective transformed program can look as follows:[9]

```
1  public static SintSarray assign(
2              __mulib__Machine__mulib__PartnerClassSarray machines,
3              SintSarray wkloads) {
4      SymbolicExecution se = SymbolicExecution.get();
5      SintSarray assignments =
6                  se.sintSarray(wkloads.length(), false);
7      for (int i = 0;Sint.concSint(i).gteChoice(wkloads.length(),se); i++) {
8          Sint wkload = wkloads.select(Sint.concSint(i), se);
9          Sint machineIndex = se.symSint();
10         __mulib__Machine machine = machines.select(machineIndex, se);
11         Sint capacity = machine.get__mulib__cap();
```

[9] Note that the program transformation is allowed to change the program into any semantically equivalent program that allows for symbolic execution of the search region method.

```
12          if (!capacity.gteChoice(wkload, se)) {
13              throw Mulib.fail();
14          }
15          machine.load(wkload, se);
16          assignments.store(Sint.concSint(i), machineIndex, se);
17      }
18      return assignments;
19  }
```

Instead of `int[]`, `SintSarray` is used (lines 1, 3 and 5) and instead of `Machine`, its *partner class* `__mulib__Machine` is used (lines 10, 11, and 15). Instead of `Machine[]`, `__mulib__Machine__mulib__PartnerClassSarray` is used (lines 2 and 10), a subclass of `PartnerClassSarray`. An instance of `SymbolicExecution` is retrieved that serves as the facade to the Mulib execution framework (see Subsection 6.3) (line 4). Aside from that, many operations have been substituted with framework operations. For instance, selecting a value from an array has been replaced by `Sarray.select(Sint, SymbolicExecution)`. Furthermore, the indicator method `Mulib.freeInt()` has been replaced to return a symbolic `Sint` (line 9). Note that, in some instances, variable types, such as `int i` (line 7), do not need to be transformed. Instead, `i` is *wrapped* (lines 7, 8, and 16).

A brief description of the program transformation can be found in Winkelmann and Kuchen [52]. In contrast, in the following, technical details are supplied and additions, such as free arrays, are included. Initially, the program transformation has been implemented using ASM [9], relying on a low-level representation of Java class files and the stack machine to perform the transformation on the level of bytecode [52]. This implementation has since been replaced by a higher-level representation in the form of a register machine using the Jimple format of Soot [47]. In fact, the program transformer can be implemented in any way that enables the generation of partner classes and could also consist of parsing text instead of processing bytecode. However, the transformations of instructions within a method (see Subsubsection 5.4.2) have been implemented and are explained in terms of Java bytecode since it is the most fundamental format. In consequence, aforementioned code is a decompiled version since the program transformation works on Java bytecode.

To abstract from the concrete framework used for the transformation, in the following the symbol `T` denotes the framework's representation of a class or interface. `F` represents fields while `M` represents a method. `I` is the framework-specific representation of an instruction. This instruction can either be a bytecode instruction or a more complex statement, such as Soot's Jimple statements [47]. `L` represents local variables while `V` represents any value. In consequence, `L` is a subtype of `V`, the latter of which can also represent values that are not first stored in a variable, i.e., values that are on the stack. In the following, a list of instructions is assumed for the transformation. Inspired by Soot's Jimple format, it is assumed that objects representing, e.g., method calls have a direct reference to values used within them. If the framework, for instance, transforms program code based on an Abstract Syntax Tree (AST) and does not offer a concrete representation for, e.g., values on the stack, adapters must be written for collecting and carrying this information, serving as the V for the framework.

5.2 Transformation Loop

As the main access point of `MulibTransformer`, to transform a class to its partner class, the procedure depicted in Algorithm 1 is called. This algorithm is implemented in the `AbstractMulibTransformer`, which implements the template pattern [27]. The algorithm combines several considerations during transformation that are elaborated in the following.

Multiple classes are passed to it which then are added to a queue (lines 1 and 2). Thereafter, classes are polled from this queue until the queue is empty (lines 3–5). The `transformClass(...)`

method is called for each of them (see Subsection 5.3).

After transforming each class, additionally some instructions are replaced and methods are enhanced. For this, we take the framework-specific representation of transformed classes and interfaces and replace each field access within a method with a specialized method (line 7, see Subsection 5.5 for an explanation of this method). This is done to account for lazy initialization or for symbolic aliasing where the fields must not be accessed directly any longer. Additionally, all methods receive a call to `PartnerClass.__mulib__nullCheck()` (line 8), again to account for symbolic aliasing and ensure that the current object is not a symbolic alias of `null`. This includes the special methods for accessing a field.

Algorithm 1: Abstract procedure for transforming the classes in a search region.

```
1  transformAndLoadClasses(Class... toTransform) : void
2      this.toTransform.addAll(toTransform);
3      while !this.toTransform.isEmpty() do
4          Class next = this.toTransform.poll();
5          transformClass(next);
6      for (_, T classNode) ∈ this.transformedClassNodes.entrySet() do
7          replaceFieldInsnsInMethods(classNode);
8          generateNullChecksForMethods(classNode);
9      Map<String, T> specialArrayTypes = getSpecialArrayTypes();
10     this.transformedClassNodes.putAll(specialArrayTypes);
11     for (String name, T classNode) ∈ this.transformedClassNodes do
12         // Writing the class to a file is abstracted from
13         Class c = loadClassForName(classNode.name());
14         this.transformedClasses.put(name, c);
```

After this, the array types used in the search region must be accounted for. While calling `transformClass(...)` (line 5), the set of array types that are used in the search region must be collected as well. It is a requirement that, when transforming the types of array values, specialized array partner classes are generated. This is discussed in more detail in Subsections 5.4 and 5.6. This set of array types is retrieved and stored in a map of pairs of the original name of the type and the framework representation of the transformed type (lines 9 and 10).

Finally, the pairs are loaded using a class loader (line 13). For this, the framework representation must be transformed to a Java class. Frameworks such as ASM and Soot offer methods for translating a type T into a byte array that can be used to define a class using `Class.defineClass(String, byte[], int, int)` [33] or to write the class to the file system so that a class loader can load it. The resulting partner class is stored in a map of pairs of the original name of the type to the partner class (line 14). Thereafter, a partner class can be retrieved from the `MulibTransformer` by using the name of the class for which we want to access the partner class.

5.3 Class Transformations

An implementation of `MulibTransformer` must transform a class or interface, alongside its (instance) fields, methods, and type hierarchy. Algorithm 2 summarizes the most important steps.

First, it is checked whether the class has already been transformed, is being transformed, or should not be transformed at all (line 2). The last of the three cases is determined by a configuration setting (see Subsubsection 8.2.6). Thereafter, the framework representation of the original class, i.e., the class that shall be transformed, is retrieved (line 4). A new representation of the transformed class is constructed (line 5). The name of this partner class is the original name

enhanced with the transformation indicator `__mulib__`. Inner classes also receive this prefix. If the package is protected and new classes must not be loaded into it[10], the package also must receive the indicator `__mulib__` as a prefix. The modifiers of the partner class, e.g., whether the class is public, private, etc., are set to equal the modifiers of the original class (line 6).

Algorithm 2: Abstract procedure for transforming a single class.

```
1  transformClass(Class toTransform) : T
2      if isIgnoredTransformedOrBeingTransformed(toTransform) then
3          ... // Handling abstracted from
4      T original = getClassNodeForName(toTransform.getName());
5      T result = new T(addTransformationIndicator(toTransform.getName()));
6      result.setModifiers(original.getModifiers());
7      T originalSuperClass = toTransform.getSuper();
8      if isInterface(toTransform) then
9          result.addInterface(getClassNodeForName(PartnerClass.class.getName()));
10     else if originalSuperClass == Object.class then
11         result.setSuper(getClassNodeForName(PartnerClassObject.class.getName()));
12     else if originalSuperClass == Throwable.class then
13         result.setSuper(getClassNodeForName(PartnerClassThrowable.class.getName()));
14     else
15         result.setSuper(transformClass(superClass));
16     for T interface ∈ toTransform.getInterfaces() do
17         result.addInterface(transformClass(toTransform));
18     // Treatment of inner and outer classes abstracted from
19     for F f ∈ original.fields() do
20         F transformedField = transformField(f);
21         result.addField(f);
22     for M m ∈ original.methods() do
23         // Native method handling abstracted from
24         M transformedMethod = transformMethod(result, m);
25         result.addMethod(transformedMethod);
26     if !isInterface(result) then
27         generateAdditionalConstructorsAndMethods(result);
28         enhanceInits(result);
29         generateOrReplaceClinit(result);
30     this.transformedClassNodes.put(toTransform.getName(), result);
31     return result;
```

Thereafter, we check the super class of the transformed class (lines 7–15). If the type we transform actually is an interface, we add the `PartnerClass` interface (lines 8 and 9). If the original supertype instead is `java.lang.Object`, `PartnerClassObject` is chosen as the supertype (lines 10 and 11). If instead the supertype is `java.lang.Throwable`, `PartnerClassThrowable` is chosen (lines 12 and 13). Finally, if none of the previous cases apply, we transform the super class and then set it as the super class of the transformed class (lines 14 and 15). Moreover, each implemented interface is transformed (lines 16 and 17).

After this, each of the fields is transformed (lines 19–21), i.e., each time the type of the field is replaced by a library type. For reference types, this also includes a call to `transformClass(...)`, or adding the respective object into the queue `toTransform` (see Algorithm 1, lines 3 and 4). For example, a field `private int i;` is transformed to a field `private Sint i;`. A field `protected A a;`

[10] Such is the case for the `java.lang` package.

is transformed to a field `protected __mulib__A a;`. A field `final int[] is;` is transformed to a field `SintSarray is;`. It is important to note that, to account for lazy initialization, the `final` keyword is discarded from the partner classes' field definition.

Then, each method is transformed (lines 22-25, further described in Subsection 5.4). For native methods, configuration options can be set to describe how they should be replaced (also see Subsubsection 8.2.6).

Finally, additional framework methods are added to each non-interface partner class (line 27), and the (class-)constructors are enhanced (lines 28 and 29). The methods that must be generated are described in more detail in Subsection 5.5. Constructors are enhanced with additional safety checks (line 28): In Java, there are default values for fields [33]. If, for instance, an `int` field is never initialized explicitly, it carries the value `0`. If a reference-typed field is not initialized, the default value `null` is assumed. Since each primitive field is replaced by a reference-typed field, unexpected `NullPointerException`s might occur. Thus, `enhanceInits(...)` inserts checks ensuring that all fields are initialized before they are used. For instance, for aforementioned field `private Sint i;` the following code is added:

```
if (i == null) {
  i = Sint.concSint(0);
}
```

Thus, the semantics for default values for `Sint` fields are preserved. Existing class initializers are deleted and replaced by similar checks for static variables (line 29). More details on the reasons for deleting class initializers in partner classes are given in Subsection 5.7.

5.4 Method Transformations

The abstract procedure to transform a method is summarized in Algorithm 3. To transform a method, first, a new framework representation of the method is constructed with the same name as the original method (line 2). The modifiers of the original method, describing, for instance, whether the method is public [33], too are set to equal those of the original method (line 3). The class declaring this new method is set to the partner class constructed in Algorithm 2 (line 4). Thereafter, the method signature is transformed: The types of the parameters, the return type, as well as the declared thrown exceptions are transformed and set for the new method (lines 5–10). As has been done for transforming fields (see Algorithm 2), primitive types are transformed to `Sprimitive`s while for reference-types new partner classes are generated (if the reference type does not correspond to a predefined type such as `SintSarray`).

If the method is abstract, no more transformation must be performed and the method representation is returned (lines 11 and 12).

Otherwise, a taint analysis is performed (line 13, see Subsubsection 5.4.1). The result of the taint analysis determines which variable types and operations are substituted using the program transformation. For all local variables of the method, it is checked whether this local variable is tainted. If it is tainted, its type is replaced analogously to how fields are transformed (lines 14–17, see Subsection 5.3). Non-array reference-typed local variables are transformed regardless of their taint value. Whether they are tainted or not is dependent on whether their type should be transformed or not, which is subject to a configuration setting (lines 18 and 19, see Subsubsection 8.2.6). Note that arrays that are not tainted can remain usual Java objects, i.e., an array `A[][]` does not need to be a subtype of `SarraySarray` but can be transformed to a `__mulib__A[][]`. If the local is neither tainted nor a reference value, it can simply be reused (lines 20 and 21).

Algorithm 3: Abstract procedure for transforming a single method.

```
1  transformMethod(T declaringClass, M m) : M
2      M result = new M(m.name());
3      result.setModifiers(m.modifiers());
4      result.setDeclaringClass(declaringClass);
5      List<T> transformedParameterTypes = transformTypes(m.parameterTypes());
6      T transformedReturnType = transformType(m.returnType());
7      List<T> transformedExceptionTypes = transformTypes(m.thrownExceptionTypes());
8      result.setParameterTypes(transformedParameterTypes);
9      result.setReturnType(transformedReturnType);
10     result.setThrownExceptions(transformedExceptionTypes);
11     if m.isAbstract() then
12         return result;
13     TaintAnalysis ta = analyzeTaint(m);
14     for L local ∈ m.locals() do
15         L transformedLocal;
16         if ta.isTainted(local) then
17             transformedLocal = transformLocal(local);
18         else if hasReferenceType(local) then
19             transformedLocal = transformToPartnerClassType(local);
20         else
21             transformedLocal = local;
22         result.addLocal(transformedLocal);
23     L seLocal = new L(SymbolicExecution.class);
24     result.addLocal(seLocal);
25     result.addInstructions(getSymbolicExecutionInsns(seLocal));
26     for I insn ∈ m.instructions() do
27         if ta.isTaintedInsn(insn) then
28             List<I> transformedInsn = transform(ta, insn);
29             result.addInstructions(transformedInsn);
30     // Treatment of try-catch blocks abstracted from
31     return result;
```

Each transformed method will begin with a statement `SymbolicExecution se = SymbolicExecution.get()`. Since `SymbolicExecution` is the facade to Mulib's search framework (see Subsection 6.3), each method must be capable to use it to contact said search framework. In consequence, a new local variable is introduced, the type of which is `SymbolicExecution` (lines 23 and 24). The static method call is inserted as the first instruction of the new method (line 25).

Thereafter, all instructions of the original method are iterated (lines 26–29). An object of the framework-specific `TaintAnalysis` is used to determine if an instruction is tainted and how it should be substituted, alongside its values. The criteria that, in addition to the taint analysis, determine whether a bytecode instruction should be substituted and what it should be substituted with are enlisted in the Subsubsections 5.4.2 and 5.4.3.

Finally, the transformed method is returned (line 31).

5.4.1 Taint Analysis

Mulib's program transformation uses an intra-procedural taint analysis to determine which variables can have symbolic values and which operations consume such values so that those opera-

tions and value types are substituted [52].

In Mulib's terminology, if a value can be symbolic it is tainted. It is tainted if it belongs to initially tainted values (such as values retrieved from fields or values created using indicator methods or is the result of a tainted instruction. An instruction is tainted if it uses any tainted values. A primitive value that is not tainted itself but is used in a tainted instruction must be *wrapped* to, e.g., an instance of `ConcSint` (see Section 4.1). Moreover, a local variable is tainted if it can contain a symbolic value, i.e., if a tainted value is stored within it. Consider the following scenario:

```
int i = 42;
int isym = Mulib.freeInt();
int iresult = i + isym;
```

In this scenario, `isym` is tainted, as it is created using an indicator method. `iresult` is tainted, because it is calculated using a tainted value. `i` is not tainted since it is concrete. Instead, `i` must be wrapped via `Sint.concSint(i)` so that it can be added to `isym` using `Sint.add(Sint, SymbolicExecution)`. Adding an assignment such as `i = isym + 1;` would taint the variable `i` as well. In this case, `i` would not be wrapped and instead receive the type `Sint`.

It is sufficient for the taint analysis to determine which values are tainted. Then, this information can be used in conjunction with the substitution criteria specified in Subsubsections 5.4.2 and 5.4.3 to determine whether an instruction is tainted (see line 27 in Algorithm 3) and thus should be substituted. Analogously, it also is not needed to determine which values must be wrapped as this can be calculated from the fact that an untainted value is used in a tainted instruction: Before executing the tainted instruction, the untainted value must be wrapped. While the current implementation of a taint analyzer using Soot also holds information on whether a value should be wrapped or not explicitly, for the sake of conciseness, this is not depicted in the following.

Algorithm 4 summarizes the procedure for the intra-procedural taint analysis. In a first step we gather all those values that are seeds of potential symbolic values (lines 3–23). In Mulib, for simplicity, it is assumed that all arguments to methods potentially are symbolic. Thus, the method `addToTainted(...)` adds all those parameter locals to a set of tainted values, for which their type is either primitive or a an array type (lines 3 and 4). Non-array reference-typed values never need to be tainted. Instead, their type can be transformed to their respective partner class type. Whether or not a reference-type should be transformed is subject to a configuration option (see Subsubsection 8.2.6). In a search region, there never are both the original class and the framework class. Similarly, all values that are retrieved from fields (lines 6–8) are tainted. Thereafter, values stored into fields are treated (lines 9–11): Primitive values that are stored in fields do not have to be tainted, but are wrapped. In contrast to primitive values, arrays cannot be wrapped. Instead, arrays must be tainted. The reason for this is that arrays can be mutated and might be represented for the constraint solver (recall Subsubsection 4.2.2). Analogously, values returned from this method are not tainted if they are not array-typed values (lines 13–15). Finally, since it is assumed that all methods accept `Sprimitive`s and `PartnerClass`es, the arguments to a method call must either be wrapped, or, in the case of an array, tainted. To do this, all method call instructions, accounting for $invokevirtual$, $invokespecial$, $invokestatic$, $invokeinterface$, and $invokedynamic$, are iterated over (lines 16–23). If, due to a configuration setting, the declaring class of a method should not be transformed, the method call does not affect the taint information (line 17). Again, the parameter values are tainted if they are array types (lines 18 and 19). Additionally, if there is a return value, this return value is tainted (lines 20–22), since each value returned by a partner classes' method is of the type of either `Sprimitive` or `PartnerClass`.

Algorithm 4: Abstract procedure for a taint analysis for determining potentially symbolic values in a single method.

```
1  analyzeTaint(M m) : TaintAnalysis
2      // Initial taint: locals
3      List<L> parameterLocals = m.parameterLocals();
4      addToTainted(parameterLocals);
5      // Initial taint: fields
6      List<I> getFieldInsns = getFieldInsns(m);
7      List<V> fieldValues = getValuesUsedByInsns(getFieldInsns);
8      addToTainted(fieldValues);
9      List<I> putFieldInsns = putFieldInsns(m);
10     List<V> putFieldValues = getValuesUsedByInsns(putFieldInsns);
11     addArrayTypedValuesToTainted(putFieldValues);
12     // Initial taint: returned values
13     List<I> returns = getReturnInsns(m);
14     List<V> returnedValues = getValuesUsedByInsns(returns);
15     addArrayTypedValuesToTainted(returnedValues);
16     for I methodCall ∈ getMethodInsns(m) do
17         if !isOfIgnoredClass(methodCall) then
18             List<V> parameterValues = getParametersOfMethodCall(methodCall);
19             addArrayTypedValuesToTainted(parameterValues);
20             V returnValue = getReturnValueOfMethodCall(methodCall);
21             if returnValue != null then
22                 addToTained(returnValue);
23         // Concretization and generalization abstracted from
24     // Propagate initial taint
25     boolean changed = true;
26     while changed do
27         changed = false;
28         for I insn ∈ getInsns(m) do
29             if !isMethodInsn(insn) && usesTaintedValues(insn) then
30                 List<V> vals = getValuesUsedByInsns(insn);
31                 changed = addArrayTypedValuesToTainted(vals) || changed;
32                 V produced = getValueProducedByInsn(insn);
33                 changed = addToTainted(produced) || changed;
34     return new TaintAnalysis(this.getTaintedValues());
```

There also is the option to treat inputs to methods that are not transformed. For instance, the inputs to this method can be labeled prematurely so that they have a valid type. This option is not discussed in the following (line 23).

After establishing the set of initial tainted values, the taint is propagated within the method. For this, all non-method call instructions are iterated over repeatedly (lines 25–33).[11] All those values used by a certain instruction are collected and, if they are arrays, added to the set of tainted values (lines 30 and 31). All values produced by tainted instructions are added to the set of tainted values (lines 32 and 33). If a new value is added to the set of tainted instructions, both `addArrayTypedValuesToTainted(...)` and `addToTainted(...)` return `true`. Thus, it is possible to determine if any new values were added and the outer loop should be run again (line 26).

The output of Algorithm 4 is an object that contains a collection of all tainted values and that can use this information to also determine whether an instruction is tainted. This can then be

[11] In practice, the outer loop is executed few times. Mostly, three iterations are performed.

used to determine tainted instructions in Algorithm 3 and decide on how the transformation of the instruction should be performed, based on the criteria in Subsubsections 5.4.2 and 5.4.3.

5.4.2 Replacement of Bytecode Instructions

The replacement of bytecode instructions is based on the result of the taint analysis described in Subsubsection 5.4.1. In all instances the bytecode instructions that should be checked to be transformed are enlisted. A small explanation of what the bytecode instructions do is given and the criteria for whether the transformation must be applied is stated. Thereafter, the transformation is described. In the following, array references are denoted tainted if

1. they are part of the parameter list
2. they are loaded from or stored in a field,
3. a tainted index value is used to store a value in them or load a value from them,
4. if they are the return value of a transformed method,
5. or if one of their variables, potentially storing the array reference, is tainted.

This corresponds to the outcome of the procedure of Algorithm 4 that is used in Algorithm 3. For a list of all bytecode instructions, Lindholm et al. [33] should be consulted.

1. $\{aaload, baload, caload, daload, faload, iaload, laload, saload\}$: Loads a value from an array using an index. There is a bytecode instruction for each array type with primitive typed-elements. This transformation must be applied if the array reference is tainted. The program transformation must either replace this bytecode instruction with $invokevirtual$ `Sarray.select(Sint, SymbolicExecution)` or with the concrete subclasses' method, such as `SintSarray.select(Sint, SymbolicExecution)`. If the former option is chosen, the returned value must be casted to the correct element-type of the `Sarray`. If the index is not tainted, it must be wrapped.

2. $\{dload, fload, iload, lload\}$: Loads a local variable or a method parameter. This transformation must be applied if the loaded variable is tainted or wrapped. The program transformation must replace it with $aload$, since we load a `Sprimitive`.

3. $\{aastore, bastore, castore, dastore, fastore, iastore, lastore, sastore\}$: Stores a value in an array using an index. There is a bytecode instruction for each array type with primitive-typed elements. This transformation must be applied if the array reference is tainted. The program transformation must either replace this bytecode instruction with $invokevirtual$ `Sarray.store(Sint, Substituted, SymbolicExecution)` or with the concrete subclasses' method, such as `SintSarray.store(Sint, Sint, SymbolicExecution)`. If the index is not tainted, it must be wrapped. If the value is not tainted and is primitive, it must be wrapped.

4. $\{dstore, fstore, istore, lstore\}$: Stores a value in a local variable or a method parameter. This transformation must be applied if the stored variable is tainted. The program transformation must replace it with $astore$, since we store a `Sprimitive`. If the value that is stored is not tainted, it must be wrapped.

5. $iinc$: Increments an `int` variable by some constant. This transformation must be applied if the variable is tainted. It must be tainted if the increment is tainted. The program transformation must replace this bytecode instruction with a sequence of bytecode instructions; - an abstract procedure is given by these three steps: (1) $aload < index >$ (2) $invokevirtual$ `Sint.add(Sint, SymbolicExecution)`, (3) $astore < index >$. The argument to the `Sint` parameter is the increment. If the increment is not tainted, it must be wrapped.

6. $getfield$: Returns the value stored in the field of an object. This transformation must be applied if the class of the object containing this field is transformed and the field's type is not an ignored reference-type or an array type with an ignored reference-type. Note that the transformation is applied in Algorithm 1, and not in Algorithm 3. The program transformation must replace this instruction using a specific getter-method (see Subsection 5.5). This step is performed automatically, if the implementing transformer is a subclass of `AbstractMulibTransformer`. Subclasses of `AbstractMulibTransformer` still must implement the respective generation process of the specific getter-methods.

7. $putfield$: Stores the value into the field of an object. This transformation must be applied if the class of the object containing this field is transformed and the field's type is not an ignored reference-type or an array type with an ignored reference-type. Note that the transformation is applied in Algorithm 1, and not in Algorithm 3. If the value put into the field is not tainted, it must be wrapped. The program transformation must replace this instruction using a specific setter-method (see Subsection 5.5). This step is performed automatically, if the implementing transformer is a subclass of `AbstractMulibTransformer`. Subclasses of `AbstractMulibTransformer` still must implement the respective generation process of the specific setter-methods.

8. $getstatic$: Returns the value stored in a static field of a class. This transformation must be applied if the class of the object containing this field is transformed and the field's type is not an ignored reference-type or an array type with an ignored reference-type. The program transformation must replace this instruction by $invokvirtual$ `SymbolicExecution.getStaticField(String)`. The `String` parameter must be the concatenation of the fully-qualified class name and the field name. For instance, when accessing field `f` of a class `A` from package `a.b.c`, the `String` `"a.b.c.A.f"` must be passed as an argument. Thereafter, a $cast$ must be added, casting the retrieved value to the search region type of the static field.

9. $putstatic$: Stores the value into a static field of a class. This transformation must be applied if the class of the object containing this field is transformed and the field's type is not an ignored reference-type or an array type with an ignored reference-type. If the value put into the field is not tainted, it must be wrapped. The program transformation must replace this instruction by $invokvirtual$ `SymbolicExecution.setStaticField(String, Object)`. The `String` parameter must be the concatenation of the fully-qualified class name and the field name. For instance, when accessing field `f` of a class `A` from package `a.b.c`, the `String` `"a.b.c.A.f"` must be passed as an argument. The argument to the `Object` is the value that shall be stored.

10. $\{d2f, d2i, d2l, f2d, f2i, f2l, i2b, i2c, i2d, i2f, i2l, i2s, l2d, l2f, l2i\}$: Cast a primitive-typed value to another type, e.g., cast a `double` to a `float`. This transformation must be applied if the value that is casted is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.d2f(SymbolicExecution)`, `Sdouble.d2i(SymbolicExecution)`, `Sdouble.d2l(SymbolicExecution)`, `Sfloat.f2d(SymbolicExecution)`, `Sfloat.f2i(SymbolicExecution)`, `Sfloat.f2l(SymbolicExecution)`, `Sint.i2b(SymbolicExecution)`, `Sint.i2c(SymbolicExecution)`, `Sint.i2d(SymbolicExecution)`, `Sint.i2f(SymbolicExecution)`, `Sint.i2l(SymbolicExecution)`, `Sint.i2s(SymbolicExecution)`, `Slong.l2d(SymbolicExecution)`, `Slong.l2f(SymbolicExecution)`, `Slong.l2i(SymbolicExecution)`}, respectively.

11. $\{dadd, fadd, iadd, ladd\}$: Adds two numbers that are either `double`s, `float`s, `int`s, or `long`s. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.add(Sdouble, SymbolicExecution)`, `Sfloat.add(Sfloat, SymbolicExecution)`, `Sint.add(Sint, SymbolicExecution)`, `Slong.add(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

12. $\{dsub, fsub, isub, lsub\}$: Subtracts one number from the other number. Both are either `double`s, `float`s, `int`s, or `long`s. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by

$invokevirtual$ {`Sdouble.sub(Sdouble, SymbolicExecution)`, `Sfloat.sub(Sfloat, SymbolicExecution)`, `Sint.sub(Sint, SymbolicExecution)`, `Slong.sub(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

13. $\{ddiv, fdiv, idiv, ldiv\}$: Divides one number with the other number. Both are either `double`s, `float`s, `int`s, or `long`s. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.div(Sdouble, SymbolicExecution)`, `Sfloat.div(Sfloat, SymbolicExecution)`, `Sint.div(Sint, SymbolicExecution)`, `Slong.div(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

14. $\{dmul, fmul, imul, lmul\}$: Multiplies two numbers. Both are either `double`s, `float`s, `int`s, or `long`s. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.mul(Sdouble, SymbolicExecution)`, `Sfloat.mul(Sfloat, SymbolicExecution)`, `Sint.mul(Sint, SymbolicExecution)`, `Slong.mul(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

15. $\{dneg, fneg, ineg, lneg\}$: Negates one number. This transformation must be applied if the value is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.neg(SymbolicExecution)`, `Sfloat.neg(SymbolicExecution)`, `Sint.neg(SymbolicExecution)`, `Slong.neg(SymbolicExecution)`} respectively.

16. $\{drem, frem, irem, lrem\}$: Computes the modulo of two numbers. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.mod(Sdouble, SymbolicExecution)`, `Sfloat.mod(Sfloat, SymbolicExecution)`, `Sint.mod(Sint, SymbolicExecution)`, `Slong.mod(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

17. $\{dcmp, fcmp, lcmp\}$: Compares two numbers. If the first value is larger, 1 is pushed onto the stack, if the second value is larger, -1 is pushed onto the stack, else 0 is pushed onto the stack. The case where NaN is involved is not currently handled by Mulib. This bytecode instruction is used to reuse all of the bytecode instructions for conditional jumps using `int` values (see 27) for `double`, `float`, and `long` values. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sdouble.cmp(Sdouble, SymbolicExecution)`, `Sfloat.cmp(Sfloat, SymbolicExecution)`, `Slong.cmp(-Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped. For an optional optimization, see 27.

18. $\{iand, land\}$: Computes the bit-wise and of two numbers. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.iand(Sint, SymbolicExecution)`, `Slong.land(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

19. $\{ior, lor\}$: Computes the bit-wise or of two numbers. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.ior(Sint, SymbolicExecution)`, `Slong.lor(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

20. $\{ishl, lshl\}$: Shifts a number to the left by a number of bits specified by another number. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.ishl(Sint, SymbolicExecution)`, `Slong.lshl(Sint, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

21. $\{ishr, lshr\}$: Shifts a number to the right by a number of bits specified by another number. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.ishr(Sint, SymbolicExecution)`,

`Slong.lshr(Sint, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

22. $\{iushr, lushr\}$: Shifts a number to the right logically by a number of bits specified by another number. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.iushr(Sint, SymbolicExecution)`, `Slong.lushr(Sint, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

23. $\{ixor, lxor\}$: Computes the bit-wise xor of two numbers. This transformation must be applied if either of the two values is tainted. The program transformation must replace this instruction by $invokevirtual$ {`Sint.ixor(Sint, SymbolicExecution)`, `Slong.lxor(Slong, SymbolicExecution)`} respectively. If either one of the two values is not tainted, it must be wrapped.

24. $\{dreturn, freturn, ireturn, lreturn\}$: Returns a primitive value from a method. This transformation must always be applied since Mulib currently assumes that each return value potentially is symbolic. The program transformation must replace this instruction by $areturn$.

25. $\{goto, goto\_w\}$: Jumps to a specified instruction in the program code. The transformation must assure that the jump now is performed to the first substituting instruction, if any. This includes calls to wrap a number.

26. $\{if\_acmp, ifnonnull, ifnull\}$: Checks whether two references equal, if a reference is not null, or if a reference is null. Jumps to a specified instruction in the program code if a condition is satisfied. In any case, the transformation must assure that the jump now is performed to the first substituting instruction, if any. This includes calls to wrap a number. This transformation must be applied if the class of (one of) the checked value(s) is transformed. The program transformation must replace these instructions by $invokevirtual$ `SymbolicExecution.evalReferencesEq(Object, Object)` where the `Object` parameters are the two values that are compared. One of the values is a `null` constant in the cases of $ifnonnull$ or $ifnull$. In the case of $ifnonnull$, the result must additionally be negated. The resulting value should be used to invoke $invokevirtual$ `Sbool.boolChoice(SymbolicExecution)`.

27. $\{if\_icmpeq, if\_icmpne, if\_icmple, if\_icmplt, if\_icmpge, if\_icmpgt, ifeq, ifne, ifle, iflt, ifge, ifgt\}$: Checks whether two `int`s (don't) equal, etc. Jumps to a specified instruction in the program code if a condition is satisfied. In any case, the program transformation must assure that the jump now is performed to the first substituting instruction, if any. This includes calls to wrap a number. The transformation must be applied if the class of (one of) the checked value(s) is transformed. The program transformation must substitute these bytecode instructions with $invokevirtual$ {`eqChoice(Sint, SymbolicExecution)`, `notEqChoice(Sint, SymbolicExecution)`, `lteChoice(Sint, SymbolicExecution)`, `ltChoice(Sint, SymbolicExecution)`, `gteChoice(Sint, SymbolicExecution)`, `gtChoice(Sint, SymbolicExecution)`, `eqChoice(SymbolicExecution)`, `notEqChoice(SymbolicExecution)`, `lteChoice(SymbolicExecution)`, `ltChoice(SymbolicExecution)`, `gteChoice(SymbolicExecution)`, `gtChoice(SymbolicExecution)`} respectively. Thereafter, the program transformation must add a call to $ifne$ [33] where the input value is the return value of the method call inserted beforehand. This is done to jump in the Java bytecode accordingly. If, for instance, `Sint.ltChoice(...)` returns true, $iflt$ succeeds. In consequence, $ifne$ will jump iff $iflt$ would jump. It is also possible to replace the unary methods with calls to the binary methods where the `Sint` parameter is fixed to `ConcSint.ZERO`. Implementations of `MulibTransformer` might also directly call `Slong`, `Sfloat`, and `Sdouble` choice methods such as `Sdouble.ltChoice(Sdouble, SymbolicExecution)` etc. In Java bytecode, comparing `double`s, `float`s, and `long`s typically is done by comparing them via their respective cmp (see 17) bytecode instruction which pushes an `int` on the stack. Thereafter, the conditional jumps described in this enumeration item are applied. Directly comparing two values, e.g., `double`s, might increase the performance. Implementation of `MulibTransformer` are also strongly encouraged to directly call the respective methods of `Sbool` instead. Since, in Java bytecode, `boolean`s are represented via `int`s, without further treatment, `Sbool`s would be treated as usual `Sint`s. However, it is more efficient to push a *pure constraint*

instead of a proxy constraint such as `b == 1`. Instead, the following substitution pairs might be used:

- if_icmpeq: $invokevirtual$ `Sbool.boolChoice(Sbool, SymbolicExecution)`
- if_icmpne: $invokevirtual$ `Sbool.negatedBoolChoice(Sbool, SymbolicExecution)`
- $ifeq$: $invokevirtual$ `Sbool.negatedBoolChoice(SymbolicExecution)`
- $ifne$: $invokevirtual$ `Sbool.boolChoice(SymbolicExecution)`

To treat `Sbool`s as `Sint`s, the configuration option `MulibConfig.VALS_TREAT_BOOLEANS_AS_INTS` (see Subsection 8.2) must be set to true.

28. $newarray$: Constructs a new array of primitive-typed elements. This transformation must be applied if the reference of the constructed array is tainted. The program transformation must replace this instruction with a call to `SymbolicExecution`'s $invokevirtual$ {`sintSarray(Sint, boolean)`, `slongSarray(Sint, boolean)`, `sdoubleSarray(Sint, boolean)`, `sfloatSarray(Sint, boolean)`, `sshortSarray(Sint, boolean)`, `scharSarray(Sint, boolean)`, `sbyteSarray(Sint, boolean)`, `sboolSarray(Sint, boolean)`}, depending on the type specified in the $newarray$ instruction. If the argument to the `Sint` parameter, describing the length of the array, is not tainted, it must be wrapped. The argument to the `boolean` parameter should be `false`.

29. $anewarray$: Constructs a new array of reference-typed elements. This transformation must be applied if the variable that will hold the constructed array is tainted. The program transformation must replace this instruction with a call to $invokevirtual$ {`partnerClassSarray(Sint, Class, boolean)` `sarraySarray(Sint, Class, boolean)`} of the class `SymbolicExecution`. If the argument to the `Sint` parameter is not tainted, it must be wrapped. The argument to the `Class` parameter should be the element type as an array type. The base type should be the transformed type. For instance, for a `int[][]`, `Sint[].class` should be passed. The argument to the `boolean` parameter should be false. The result should be casted to the specifically generated array type (see Subsection 5.6).

30. $multianewarray$: Constructs a multi-dimensional array. This transformation must be applied if the variable that will hold the constructed array is tainted. The program transformation must substitute this instruction with a call to $invokevirtual$ `SymbolicExecution.sarraySarray(Sint[], Class)`. The argument for the `Sint[]` parameter is an array containing the dimensionalities of the initialized array. If the counts for the dimensionalities are not yet tainted, they must be wrapped and added into a new intermediate array. The result must be casted to the specific array of arrays-type.

31. $arraylength$: Returns the length of an array. This transformation must be applied if the array is tainted. The transformation must substitute this instruction with a call to $invokevirtual$ `Sarray.length()`.

32. $checkcast$: Casts the type of a reference-typed value to a specified type. This transformation must be applied for any $checkcast$ bytecode instruction in preparation for including free objects (see Dageförde, Winkelmann, and Kuchen [22]). The transformation must substitute this instruction with a call to $invokevirtual$ `SymbolicExecution.castTo(Object, Class)`. The `Object` parameter represents the value that should be casted. The `Class` parameter is the class to which to cast. The returned value of this method ust be casted to the appropriate type using $checkcast$.

33. $instanceof$: Returns `true` if the given value has the specified type. This transformation must be applied for any $instanceof$ bytecode instruction in preparation for including free objects (see Dageförde, Winkelmann, and Kuchen [22]). The `Object` parameter represents the value that should be evaluated. The `Class` parameter is the class for which it is checked whether the value is an instance of it. The transformation must substitute this instruction with a call to $invokevirtual$ `SymbolicExecution.evalInstanceof(Object, Class)`.

34. $\{invokedynamic, invokeinterface, invokespecial, invokestatic, invokevirtual\}$: Invokes a method. This transformation only applies if the invoked method is not an indicator method (see Subsubsection 5.4.3). If the invoked method is an indicator method, a specialized transformation is conducted. Otherwise, this transformation must be applied if the class containing the invoked method is transformed. The new method of the partner class should be invoked instead. All parameters that are not tainted must be wrapped.

35. $\{lookupswitch, tableswitch\}$: Evaluates a `switch` statement and jumps to the respective program portion. This instruction must be transformed if the evaluation criterion is based on tainted values. The program transformation must treat a $switch$ bytecode instruction as a sequence of `if` statements (see 27).

5.4.3 Indicator Methods

There are also several indicator methods that must be treated as special cases during transformation. They are enlisted and explained in the following. All methods are static methods in the class `Mulib`. If they are not transformed, most of them will throw an error indicating that they should have been substituted using a program transformation.

1. `fail()`: Discards the currently explored `ChoiceOption` (see Subsection 6.1). The search tree is pruned so that no descendants of this choice option are explored. This method should not be transformed and kept as-is.

2. {`freeDouble()`, `freeFloat()`, `freeInt()`, `freeLong()`, `freeShort()`, `freeChar()`, `freeByte()`, `freeBool()`}: These methods should generate a new symbolic value of the specified type. They should be substituted by
$invokevirtual$ {`symSdouble()`, `symSfloat()`,
`symSint()`, `symSlong()`,
`symSshort()`, `symSchar()`,
`symSbyte()`, `symSbool()`} of `SymbolicExecution`.

3. {`freeDouble(double, double)`, `freeFloat(float, float)`, `freeInt(int, int)`, `freeLong(long, long)`, `freeShort(short, short)`, `freeChar(char, char)`, `freeByte(byte, byte)`}: These methods should generate a new symbolic value of the specified type within a specified lower and upper bound. They should be substituted by
$invokevirtual$ {`symSdouble(Sdouble, Sdouble)`, `symSfloat(Sfloat, Sfloat)`,
`symSint(Sint, Sint)`, `symSlong(Slong, Slong)`,
`symSshort(Sshort, Sshort)`, `symSchar(Schar, Schar)`,
`symSbyte(Sbyte, Sbyte)`} of `SymbolicExecution`. If the arguments to the parameters are not tainted, they must be wrapped.

4. `freeObject(Class)`: This method generates a new symbolic object of a class of the specified type. To generate an instance of `A`, `A a = freeObject(A.class);` should be specified. If the original `Class` parameter is not an array, this method should be substituted by $invokevirtual$ `symObject(Class)` where the `Class` parameter is the transformed class that should be initialized. This method is also used to initialize free/symbolic arrays, i.e., arrays with free elements and a free length. Thus, if the `Class` parameter is an array, the method call should be substituted by one of the following method calls defined by `SymbolicExecution`:
$invokevirtual$ {`sintSarray(Sint, boolean)`, `slongSarray(Sint, boolean)`,
`sdoubleSarray(Sint, boolean)`, `sfloatSarray(Sint, boolean)`,
`sshortSarray(Sint, boolean)`, `scharSarray(Sint, boolean)`,
`sbyteSarray(Sint, boolean)`, `sboolSarray(Sint, boolean)`,
`partnerClassSarray(Sint, Class, boolean)`, `sarraySarray(Sint, Class, boolean)`}.
In consequence, `freeObject(int[].class)` should be substituted with `sintSarray(Sint, boolean)`. The `Sint` parameter should be initialized with a new call to

invokevirtual `SymbolicExecution.symSint()`. This parameter controls the length of the array. The `boolean` parameter should be fixed to true. For `partnerClassSarray(Sint, Class, boolean)` and `sarraySarray(Sint, Class, boolean)`, the `Class` parameter should be set to the array's component type as a typical Java type. For instance, for a new symbolic array of the original type `A[][]`, `__mulib__A[]` is passed.

5. `freeArray(int, Class)`: Proceed analogously to the array-case of 4. However, the length is not set to a symbolic value but to the `int` parameter. If the argument to the parameter is not tainted, it must be wrapped.

6. {`remember(double, String)`, `remember(float, String)`, `remember(int, String)`, `remember(long, String)`, `remember(short, String)`, `remember(char, String)`, `remember(byte, String)`, `remember(boolean, String)`, `remember(Object, String)`}: Creates a snapshot of the value and remembers it for labeling using a name specified as a `String`. All method calls to these indicator methods should be replaced by *invokevirtual* `SymbolicExecution.nameSubstitutedVar(Substituted, String)`. If primitive values are remembered and are not tainted, they must be wrapped. Note that the indicator methods specified in 2, 4, and 3 also have a shortcut for initializing and immediately remembering the values. The respective indicator method's name have a *remembered* prefix and an additional `String` parameter representing the name, e.g., `Mulib.rememberedFreeInt(String)`. The respective substitution methods' names have a *named* prefix and an additional `String` parameter representing the name, e.g., `SymbolicExecution.namedSymSint(String)`.

7. `assume(boolean)`: Pushes a new constraint onto the constraint stack without adding a choice point. Currently, the choice point is created before accessing `assume(...)`, for reasons explained in Subsubsection 4.1.2. The constraint is the passed `boolean` argument. The method should be substituted by a call to *invokevirtual* `SymbolicExecution.assume(Sbool)`. If the argument to the method is not tainted, it should be wrapped.

8. `check(boolean)`: Checks the current constraint stack with a new constraint. The new constraint is not pushed onto the constraint stack and no choice point is generated. The constraint is the passed `boolean` argument. The method should be substituted by a call to *invokevirtual* `SymbolicExecution.check(Sbool)`. If the argument to the method is not tainted, it should be wrapped.

9. `checkAssume(boolean)`: Checks the current constraint stack with a new constraint. The new constraint is only pushed onto the constraint stack if the constraint stack and the new constraint are satisfiable. The constraint is the passed `boolean` argument. The method should be substituted by a call to *invokevirtual* `SymbolicExecution.checkAssume(Sbool)`. If the argument to the method is not tainted, it should be wrapped.

10. `isInSearch()`: Returns true, if search is currently conducted. This is useful when building model classes that might generate new values during search, e.g., symbolic databases. The method should be substituted by a call to *invokevirtual* `SymbolicExecution.isInSearch()`.

11. {`pickFrom(double[])`, `pickFrom(float[])`, `pickFrom(int[])`, `pickFrom(long[])`, `pickFrom(short[])`, `pickFrom(char[])`, `pickFrom(byte[])`, `pickFrom(boolean[])`, `pickFrom(Object[])`}: Returns a new value which is a symbolic alias (see Winkelmann and Kuchen [53]) of one of the values in the respective array. The transformation should replace the call with a call to either
invokevirtual {`aliasOf(Sdouble[])`, `aliasOf(Sfloat[])`, `aliasOf(Sint[])`, `aliasOf(Slong[])`, `aliasOf(Sshort[])`, `aliasOf(Schar[])`, `aliasOf(Sbyte[])`, `aliasOf(Sbool[])`, `aliasOf(PartnerClass[])`}, if the input array is not tainted, or otherwise
invokevirtual {`aliasOf(SdoubleSarray)`, `aliasOf(SfloatSarray)`, `aliasOf(SintSarray)`, `aliasOf(SlongSarray)`, `aliasOf(SshortSarray)`, `aliasOf(ScharSarray)`, `aliasOf(SbyteSarray)`, `aliasOf(SboolSarray)`, `aliasOf(PartnerClassSarray)`} of `SymbolicExecution`.

5.5 Additions to Partner Classes

By extending either `PartnerClassObject` or `PartnerClassThrowable`, the search region classes inherit a set of methods that embed them in the routines of Mulib. Still, some methods must be generated as they are specific to the content of the object and would become inefficient using reflection. Thus, each non-interface class receives a set of additional methods that must be generated by the implementation of `MulibTransformer`. These methods are enumerated in the following for a transformed class `A`:

1. `A` must implement a public constructor `A(SymbolicExecution)`. This constructor prepares the lazy initialization of the object. In this constructor, the super constructor with the same parameter signature is called. If the super class is `PartnerClassObject` or `PartnerClassThrowable`, the empty constructor is called instead. If the passed argument for `SymbolicExecution` is not null, $invokeinterface$ `PartnerClass.__mulib__setDefaultIsSymbolic()` is called.

2. `A` must receive a public constructor `A(Object, MulibValueTransformer)`. This constructor is used for transforming the original object (the first parameter) into its search region representation. The various fields should be transformed using `MulibValueTransformer`. If both the original class and the search region class are in the same module and package, protected fields can be accessed immediately. Otherwise, reflection must be used to access the fields. Potentially, if the original class is not visible from the search region class, reflection must be used to retrieve the class and the fields of the class as well. The transformation constructor of the super class should also be called. For those classes that have `PartnerClassObject` or `PartnerClassThrowable` as their supertypes, the empty constructor should be called.

3. `A` must receive a public constructor `A(A, MulibValueCopier)`. This constructor is used for copying a `PartnerClass` value (the first parameter) using `MulibValueCopier`. The copy constructor of the super class should also be called.

4. `A` must receive a public method `PartnerClass.copy(MulibValueCopier)`. This method creates a new object and initializes it using the constructor from 3.

5. `A` must receive a public method `PartnerClass.__mulib__getOriginalClass()`. This method returns the original class of the search region class. If the original class is not visible, reflection must be used.

6. `A` must receive a public method `PartnerClass.__mulib__blockCacheInPartnerClassFields()`. This method calls $invokeinterface$ `PartnerClass.__mulib__blockCache()` for all fields with partner class-typed elements. The method should also be called for the super class.

7. `A` must receive a public method `PartnerClass.__mulib__initializeLazyFields(SymbolicExecution)`. The method calls the respective method initializing a field with a symbolic value. For primitive types the methods $invokevirtual$ `symSint()` etc. should be called. For reference types, $invokevirtual$ `symObject(...)` or one of the `SymbolicExecution.XYZSarray(...)` methods should be called.

8. `A` must receive a public method `PartnerClass.__mulib__getFieldNameToSubstituted()`. The method returns a `Map<String, Substituted>` which is a flat map representation of the content of this object. The result should be enriched by also adding the result of calling the super class' implementation.

Moreover, two methods are added for each field: Assume that a transformed class `A` in the package `"a.b.c"` has a field of the form `private Sint A.xyz`. The following methods must be generated:

```
1  private Sint get__mulib__xyz() {
2    if (__mulib__isToBeLazilyInitialized()) {
3      SymbolicExecution.get().initializeLazyFields(this);
4    }
```

```
5     __mulib__nullCheck();
6     if (__mulib__cacheIsBlocked()) {
7       return SymbolicExecution.get().getField(this, "a.b.c.A.xyz", Sint.class);
8     } else {
9       return this.xyz;
10    }
11  }
12
13  private set__mulib__xyz(Sint var) {
14    if (__mulib__isToBeLazilyInitialized()) {
15      SymbolicExecution.get().initializeLazyFields(this);
16    }
17    __mulib__nullCheck();
18    if (__mulib__cacheIsBlocked()) {
19      SymbolicExecution.get().putField(this, "a.b.c.A.xyz", var);
20    } else {
21      this.xyz = var;
22    }
23  }
```

The purpose of this method is to check whether we need to lazily initialize the values of an object (see Khurshid, Păsăreanu, and Visser [29] and Winkelmann, Troost, and Kuchen [55]) (lines 2–3 and 14–16). We additionally perform a null-check (line 5 and line 17) since, in the case of symbolic aliases, an initialized object might represent `null` (see Winkelmann and Kuchen [53]). Finally, if the object is already represented in the solver, i.e., it's fields should not be accessed anymore directly, we either pose a respective `PartnerClassObjectConstraint` (see Subsubsection 4.2.2) (lines 6 and 7 and lines 18 and 19) and otherwise perform the operation directly on the field (lines 9 and 21).

5.6 Special Considerations for Arrays

While transforming the search regions, an implementation of `MulibTransformer` must collect the various array types that are encountered. The reason for this is that we for each array type a new class should be generated that (implicitly) extends either `PartnerClassArray`, for arrays with a non-array component type, or `SarraySarray` otherwise. In Java, arrays can have a dimensionality of up to 255 [33]. It would be an immense overhead to create partner classes for all these arrays. For this reason, information on this should be collected and new search region representation for array types should be generated.

5.7 Special Considerations for Class Initializers and Static Fields

Mulib currently discards class initializers. Instead, when invoking a search region, the current values of static fields in the original program are transformed and used for the search region. In fact, Mulib does not use the static fields of search region classes at all. Instead, the values are collected in a separate structure, `StaticVariables`. There are multiple reasons for this. First and foremost, class initializers are only executed once, when the class is loaded [33]. If there would be symbolic values and choice points in class initializers, the process of loading a class would have to be undone during backtracking. This is a rather complex undertaking Java does not offer an API for [28]. Additionally, using complex logic in class initializers is a discouraged practice. As such, the effort does not seem worth the cost. In fact, to our knowledge, no other tool for symbolic execution supports this feature comprehensively. Moreover, having symbolic values set in class initializers would immediately imply leaking symbolic values outside of the search region. After all, any method that can access a static variable, even outside of a search region, might then retrieve a symbolic value. However, for the use-case of program verification, treating static initializers should be considered in the future. Mulib might offer a configuration option by means of which

class initializers are considered. To achieve this, static initializers can be renamed so that they are not considered to be static initializers by the JVM. Consequently, they are not executed when the class is first loaded. Instead, the moment of applying the class initializer can then be decided upon either manually, or using marker methods introduced by the program transformation.

While beforehand class initializers have been discussed, accessing static fields in Mulib works by means of an auxiliary structure, namely `StaticVariables`. The reason not to directly access static fields in the search region is that this would be detrimental for exploring the search region using multiple threads: Each thread might write to the static variable. Thus, executing the search region with multiple executors would become impossible, as each execution might overwrite the symbolic value that is assumed to be contained in a certain field. Mulib still allows to set static values to a symbolic value within the search region. This, however, is not reflected in the original class or by other executors. Instead, an executor-specific instance of `StaticVariables` is updated (see Enumeration Items 8 and 9 in Subsubsection 5.4.2). For this, during the program transformation, `MulibTransformer` must collect the potentially accessed static fields.

5.8 Differences to Muli$_{sjvm}$

The program transformation replaces the compiler [17] of Muli$_{sjvm}$. In contrast to the custom compiler, at no point bytecode is generated that is not compatible to the bytecode consumed by a standard JVM. After all, the Mulib search framework (see Section 6) is implemented entirely in Java. The task of the program transformer is solely to inject the search framework's functionality into the original class, generating partner classes. The program transformer completely relies on tooling that is delivered with a typical JDK (and third-party Java libraries). By eliminating the need for additional tooling, Mulib can be deemed more approachable by users. Moreover, since the program transformation in its current form is based on Java bytecode, it does not lose the advantages offered by Muli$_{sjvm}$: Still, symbolic execution is offered for all languages that compile to Java bytecode. This also means that existing libraries can be reused. Still, it is also possible to use alternative program transformers such as transformers based on source code. Mulib's search framework is orthogonal to the chosen program transformer.

The input as well as the output of the program transformation again is valid Java bytecode. Hence, the custom SJVM also is not needed any longer and an off-the-shelf high-performance JVM such as HotSpot can be used directly to execute the search region, proving to be more performant [52]. As an added benefit, since the current Mulib program transformation is defined on bytecode, it also opens the door for extending several JVM languages with CLOOP. The indicator methods might as well be used to formulate search regions in, e.g., Scala, Groovy, Clojure, or Kotlin code where Mulib is used as a library. Due to this and the extensive DSL capabilties of, e.g., Scala, Mulib facilitates the employment of CLOOP even outside the scope of Muli.

The program transformation can be used in an ad-hoc manner, or alike a compiler: Either, the class files of partner classes are created during run time and then are loaded, or they can be written to disk and are loaded using the system class loader.

The fact that Mulib maintains a separate representation for code outside of a search region and inside of a search region intuitively caters well to the concept of just-in-time compilation [15] that is employed within modern JVMs. In the search region, multiple bytecode instructions, such as `iadd`, must account for more complex operations where it must be checked whether a symbolic value is involved. Outside of a search region, however, the bytecode instructions perform rather simple operations that can be compiled more efficiently than more complex operations within a search region. Performing one just-in-time-compilation that caters to both scenarios appears to be a rather hard task. Hence, having two separate representations means that the code can be just-in-time compiled for each scenario in separate. Consequently, outside of a search region, just-in-time compilation performance is not hindered by the fact that code might also be executed

symbolically. In turn, the search region code can also be just-in-time compiled.

Another upside of using a program transformation is that it is particularly easy to create *model classes*. Model classes, also sometimes referred to as a category of mock classes, must be created if some native method is used or some calculations are infeasible when executed symbolically. In this case, classes and methods are manually created that model the original behavior in a manner so that symbolic execution can proceed. For instance, to symbolically execute an application using a database, a model must be created mimicking the database's behavior (see, e.g., Winkelmann and Kuchen [54]). If the database is written in, e.g., C and native methods are used to contact it, these native methods cannot be symbolically executed. On the other hand, the functionality of a database might be connected with too much overhead that is not needed for symbolic execution. For instance, for verifying assertions in a program, it does not matter whether a database uses an index. This, in fact, might even slow down execution since more complex constraints are formulated. During the program transformation, such classes and methods can be replaced so that in the search region a model class is used instead. Mulib offers configuration options for this (see Subsubsection 8.2.6). In the model class, Mulib's indicator methods can be used to spawn new symbolic values, such as new entries in a modelled database (see, e.g., Winkelmann and Kuchen [54] where values are spawned via the interpretation of custom bytecode instructions instead of such indicator methods). Other symbolic execution tools, such as Symbolic PathFinder [29], also offer a model interface by means of which spawning symbolic values is possible. The major difference is that in Mulib, this feature is not an extension, but rather part of the core components: The program transformation transforms classes with indicator methods in any case. In Symbolic PathFinder, this feature is implemented by altering the state of an SJVM which, in turn, is the true core component for symbolic execution. The execution core of a symbolic execution engine can be expected to be better-maintained than extensions.

6 Search

The the first large component of Mulib, the program transformation (see Section 5), transforms a Java bytecode program into a new program that utilizes partner classes and search framework methods. The other large component of Mulib thus is its search mechanism. *Search* in Mulib denotes the exploration of a search tree. The search tree represents the choices encountered in a search region while running the respective program [21]. In this section, first, the structure of a search tree in Mulib is explained in Subsection 6.1. Thereafter, in Subsection 6.2, the steps for preparing to execute the search region to explore the representing search tree are described. Then, the access point by means of which the transformed method, i.e., the search region, can use the Mulib framework is detailed (Subsection 6.3). Subsequently, it is illustrated how new choices points are created, how symbolic values are initialized, and how calculations are performed (Subsection 6.4). Subsection 6.6 then illuminates how search strategies are implemented in Mulib. Again, this section closes with 6.7 providing a comparison of Muli$_{sjvm}$ and Mulib.

6.1 Search Tree

A search tree is explored while executing the search region symbolically [21]. Each time a choice point is encountered, a node is added to the tree with the choice options being the edges to subtrees. Each time a path solution is uncovered, a leaf node is added to the tree. In the following, first, the structure of the search tree is described. Figure 8 shows the search tree in form of a class diagram. It is constructed in a linked fashion: Each search tree has a root `Choice` with a single `ChoiceOption`. `Choice` here is the class representing a choice point with its options. Each intermediate node is a `Choice`. Each `Choice` has one or more `ChoiceOption`s. In other words: the exploration of the search region *forks* at a choice and options for exploring the search tree are uncovered in form of `ChoiceOption`s. Leaf nodes are either instances of `PathSolution`, representing a returned or thrown (see Figure 2) result of the search region, `Fail`s, indicating that a constraint stack is unsatisfiable or that `Mulib.fail()` has been called explicitly to prune the search tree, or `ExceededBudget`s, indicating that the search tree was pruned due to a budget (see Subsubsection 8.2.3). The root choice is the only node in the tree that does not have a `TreeNode.pEdge`, i.e., there is no edge from a parent node to this node.

This structure of the search tree makes it particularly easy to determine the deepest shared ancestor of two nodes[12], determining an existing path between two choice options, and retrieving the path from the root to a choice option. For this, it mostly suffices to get the `TreeNode.pEdge` of the parent `Choice`, for which we can then invoke the same operation. For these three operations static helper methods have been implemented in `SearchTree` that are used by the search framework. These are subsequently described when needed.

6.2 Preparing an Execution

In Mulib, a search tree is explored using one or several *executions*. In each execution, the `java.lang.invoke.MethodHandle` representing the search region is invoked anew. This is different from the previous Muli implementation in which detailed trails for backtracking the program to the state of a specific choice point are maintained [21]. Before the search region is invoked for such an execution, some preparatory steps must be performed. First, the transformed arguments to the search region must be copied (recall Figure 3). The reason for this is that, while executing the search region, input objects might be mutated. While, in the future, as remarked in work on Muli$_{sjvm}$ [16], a more elaborate copy-on-write-approach might be implemented, runtime

[12] At least the single choice option of the root choice is shared.

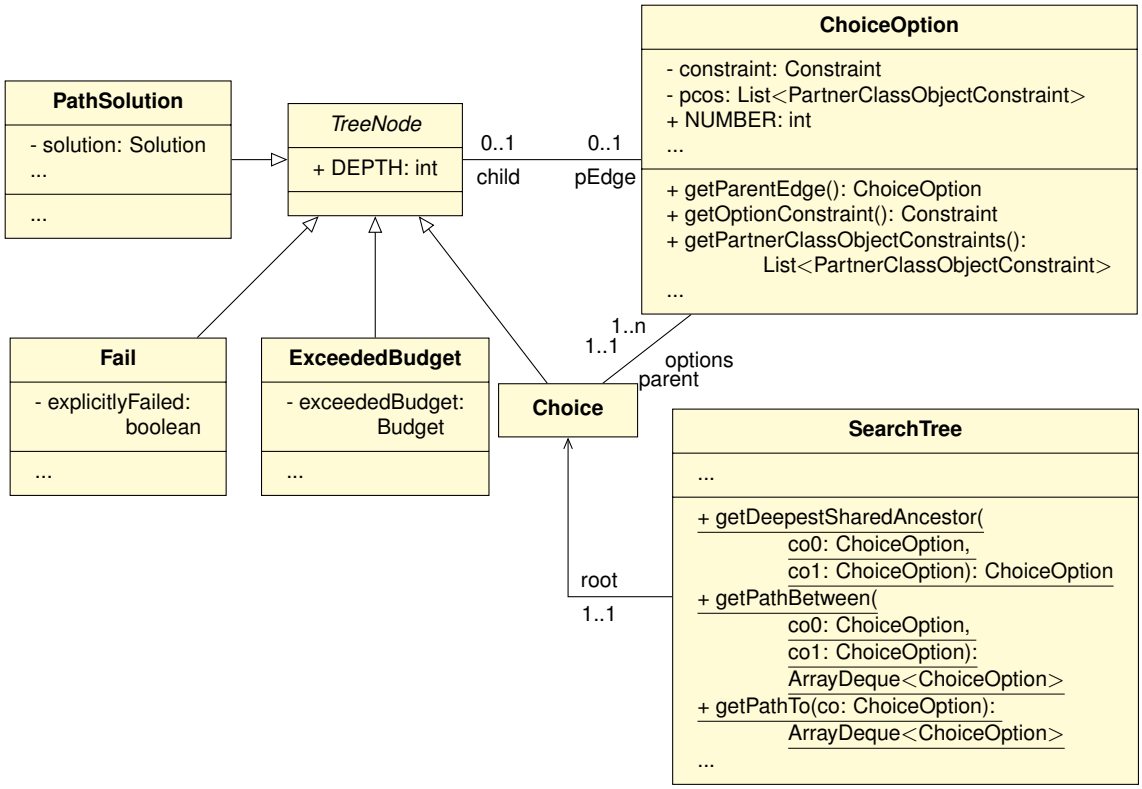

Figure 8: The classes making up the search tree.

measurements suggested that, for the encountered workloads, the overhead of performing deep copies is benign.[13] This is done in `MulibExecutor` (recall Figure 4). Additionally, an object of the class `StaticVariables` is reset. This class is used by `MulibExecutor` to manage any static fields that might be read from or stored within in the search region. The static fields are collected by the instance of `MulibTransformer` (see Subsection 5.7). As soon as a static variable is read from in Mulib, a deep copy is generated.

Moreover, before executing the search region, the next choice option to explore in the search tree must be decided upon. Initially, there only is one root choice option. However, after the search region has been executed, potentially new choices with their choice options have been encountered. Subsection 6.4 describes how new choice options are added to the search tree and how we navigate to a choice option during execution. In contrast, in this subsection it is explained what the necessary steps are for evaluating whether a new initial choice option is satisfiable before starting a new execution in Mulib.

The fundamental classes for conducting search in Mulib are depicted in Figure 9 among their most important methods and interdependencies. `MulibExecutorManager` is the instance that manages one or many `MulibExecutor`s, i.e., search strategies. When executing `MulibExecutorManager.getPathSolutions()`, a loop is started in which `MulibExecutor.getPathSolution()` is invoked repeatedly (recall Figure 4). In each iteration of this loop the `MulibExecutor` must decide on an initial choice option to which the execution should navigate. This choice is dependent on the concrete search strategy that is applied. The search strategies will be discussed in more detail in Subsection 6.6. For now, it is assumed that the search strategy has determined a choice option to navigate to.

[13] Typically, for inputs with a larger ”size” the complexity of constraints on these inputs far outweighs the cost of any deep copy.

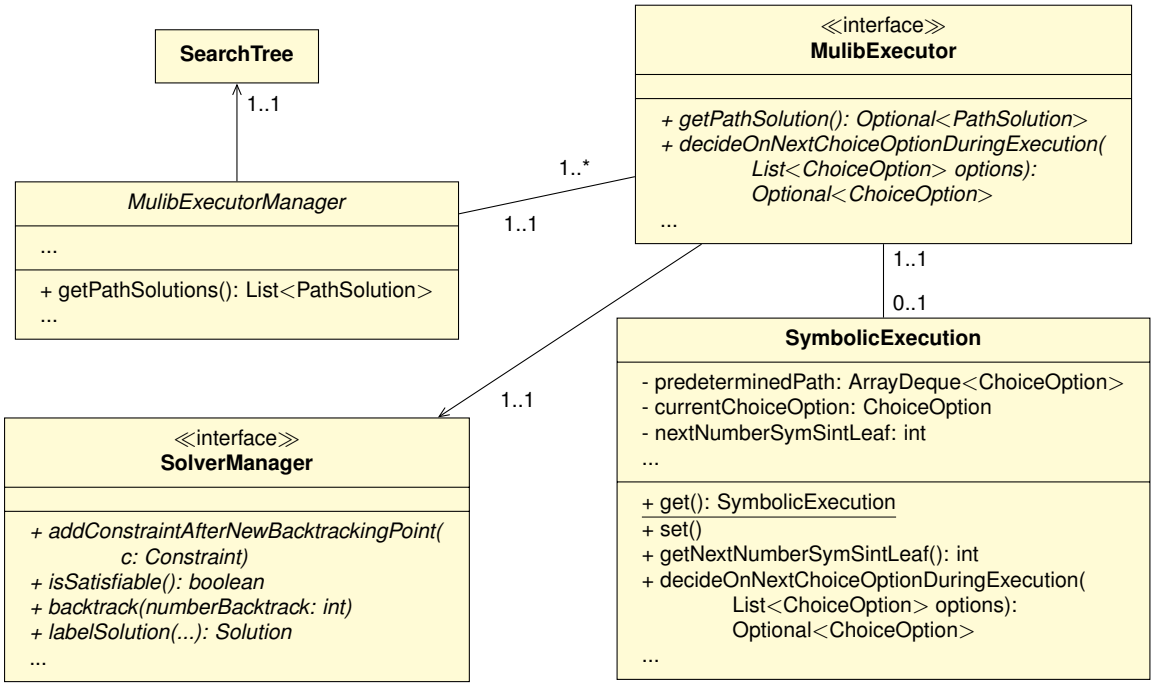

Figure 9: The fundamental classes for starting a new execution of the search region in Mulib.

Mulib is efficient in that it does not execute the search region if the constraints added for reaching the new choice option make the overall constraint system unsatisfiable. To ensure this, it must be checked whether or not the constraint stack for reaching the new choice option is satisfiable. If the constraint for this choice option makes path condition, i.e., the constraint stack, represented in the instance of `SolverManager`[14] unsatisfiable, we should not execute the search region up until this point and only then find out. Consequently, it should be assured that all constraints encountered when navigating to the choice option in the search region are on the stack of the `SolverManager` and that the constraint solver evaluates this constraint stack to be satisfiable. Only then the search region should be invoked.

Mulib was designed from the start with incremental constraint solvers in mind [52]. When evaluating a constraint stack, incremental constraint solvers *learn* lemmas so that if the constraint stack is extended, the performance of checking the satisfiability is vastly increased [34, 52]. Incremental constraint solvers can *push* scopes in which constraints are added onto a stack of scopes. They then can pop a scope and thus remove all constraints (and lemmas) that were added for it. Another way to think about scopes in incremental constraint solvers is that we can create a backtracking point for each choice option. In other words: For each encountered `ChoiceOption` we can push a scope. We can then backtrack to the `SolverManager`'s representation for previous `ChoiceOption` by popping scopes. Thus, in the following, the terminology of *creating new backtracking points* is used for pushing new scopes, and *backtracking* when these scopes are popped.

To ensure that executions are only started for valid choice options, we first adapt the constraint stack represented in the `SolverManager`. The instance of `MulibExecutor` additionally remembers the last choice option it has evaluated. A helper method of `SearchTree` (see Figure 8, `SearchTree.getDeepestSharedAncestor(...)`) is used to calculate the deepest shared ancestor of the new choice option selected for evaluation and the last evaluated choice option. In other words: The `Choice-`

[14] More details on the architecture of `SolverManagers` in Mulib can be found in Section 7.

Option that is the deepest in the tree and from which we can navigate via a path of strictly increasing depth to both the previously evaluated choice option and the newly selected choice option is calculated. Then, the difference in depth between the last evaluated choice option and this common ancestor is evaluated. We backtrack the SolverManager by this difference, i.e., we pop this number of scopes. Thereafter, the path between the newly selected choice option and the common ancestor is determined using another one of the helper methods in SearchTree (SearchTree.getPathBetween(...)). For each of the choice options on the path, SolverManager.addConstraintAfterNewBacktrackingPoint(...) is executed, i.e., the delta of all constraints is pushed to the constraint solver. We then evaluate SolverManager.isSatisfiable(). If this method invocation returns true, the choice option is satisfiable. If false is returned, the choice option is marked as unsatisfiable and is not regarded further. Note that, since each MulibExecutor has an own instance of SolverManager, this procedure is thread-safe if the selection of the next choice option is thread-safe. Hence, it can also be employed by multiple MulibExecutors in parallel.

If the choice option turns out to be satisfiable, MulibExecutor now creates a new instance of SymbolicExecution passing the ChoiceOption that should be navigated to and invokes the search region. After executing the search region and extracting a return value, the SolverManager is used to *label* the solution, i.e., assign concrete primitive values to all Sprimitives, and transform the search region representation of reference-typed values to their original representation with labeled fields.

6.3 Symbolic Execution

A new instance of SymbolicExecution is created for each new invocation of the search region. SymbolicExecution implements the facade design pattern [27] and is used in the transformed partner classes to contact the Mulib framework. When creating a new execution of the search region, the respective instance of SymbolicExecution is set in a static field. This static field is of type java.lang.ThreadLocal. ThreadLocal is a class that allows to access thread-specific instances of objects. Since each MulibExecutor is run in a separate thread the current execution's instance of SymbolicExecution can be retrieved in a thread-safe manner by calling the static method SymbolicExecution.get() which accesses said ThreadLocal.

By means of the managing instance of MulibExecutor the SymbolicExecution object can, e.g., add new constraints to the SolverManager. Furthermore, SymbolicExecution delegates all noteworthy business logic, such as the initialization of new symbolic values, the calculation of expressions and constraints (see Subsection 4.1), and evaluation of potential and creation of new choice points to the ValueFactory, CalculationFactory (see Subsection 6.5), and ChoicePointFactory (see Subsection 6.4) respectively.

An instance of SymbolicExecution furthermore has the task to keep track of execution-specific data. For some budgets that can be specified (see Subsubsection 8.2.3), it holds an ExecutionBudgetManager. Furthermore, upon its creation, SymbolicExecution extracts and stores a *predetermined path* of ChoiceOptions (recall Figure 9). For this, all ChoiceOptions that lead to the navigated-to ChoiceOption that was passed by the MulibExecutor are extracted via SearchTree.getPathTo(...). Each choice option in the search tree represents a portion of the search region that was reached after following a certain path. This portion of the search region can only be navigated to by performing a unique sequence of decisions on the choice points. In most cases, this decision is the selection of a choice option. Such a choice option represents one of the two branches of an if-condition. To reach subsequent choice options, the if-condition must be evaluated so that the navigated-to choice option can be reached. The role of SymbolicExecution.predeterminedPath thus is to fix the decision on choice options to reach the navigated-to choice option. This predetermined path is explained in more detail in Subsection 6.4. Additionally, for each of the Java primitive types, instances of SymbolicExecution carry a counter. The role of this counter is further elaborated in Subsection 6.5.

6.4 Choice Points

The procedure of Mulib for dealing with choice points has already been sketched out in previous work [52]. In the following, more technical details for this approach are given.

During the transformation described in Section 5, if a symbolic value can potentially be part of, e.g., an if-condition, bytecode instructions such as $iflt$ are substituted by method calls such as `Sint.ltChoice(...)`. There are several such choice methods. While `SymbolicExecution` is used to call the method, the call is simply forwarded to a `ChoicePointFactory`, such as the `SymbolicChoicePointFactory`. The behavior of this `SymbolicChoicePointFactory` is summarized in Algorithm 5.[15]

Algorithm 5: Simplified procedure for evaluating binary choice points.

```
1  threeCaseDistinctionTemplate(SymbolicExecution se, Constraint c) : boolean
2      if c instanceof ConcSbool then
3          return ((ConcSbool) c).isTrue();
4      ChoiceOption currentCo = se.getCurrentChoiceOption();
5      if se.transitionToNextChoiceOption() then
6          currentCo = se.getCurrentChoiceOption();
7          return currentCo.NUMBER == 0;
8      Choice newChoice = new Choice(currentCo, c, Not.newInstance(c));
9      Optional<ChoiceOption> possibleNextCo =
         se.decideOnNextChoiceDuringExecution(newChoice.getChoiceOptions());
10     if possibleNextCo.isEmpty() then
11         se.notifyNewChoice(newChoice.depth, newChoice.getChoiceOptions());
12         throw Backtrack.getInstance();
13     else
14         int coNumber = possibleNextCo.get().NUMBER;
15         int otherCoNumber = coNumber == 0 ? 1 : 0;
16         ChoiceOption notTakenCo =
             newChoice.getChoiceOptions().get(otherCoNumber);
17         se.notifyNewChoice(newChoice.depth, List.of(notTakenCo));
18         return coNumber == 0;
```

Algorithm 5 is called by each of the choice methods. The sole difference between these methods is the value of the `Constraint` parameter c that is passed to it. For instance, calling `ltChoice(...)` will pass an instance of `Lt` (recall Subsubsection 4.1.2). This algorithm distinguishes three cases. The first case (lines 2–3) is that the constraint, in fact, is concrete, i.e., no symbolic value is involved in it. In this case, Mulib will never create a choice point and instead will return the respective boolean value. If `ltChoice(Sint.concSint(3), Sint.concSint(4))` is called, the result of applying `Lt.newInstance(Sint.concSint(3), Sint.concSint(4))` is a `ConcSbool` encoding `true`. If the constraint involves a symbolic value, c contains a constraint expression. The second case (lines 4–7) is checked. The instance of `SymbolicExecution` contains a stack of predetermined `ChoiceOption`s (recall Subsection 6.3). If this path is not empty, Mulib must still force the execution to take decisions to reach the targeted choice option in the search tree. Thus, it is evaluated whether we still are on the predetermined path (line 5). If there is a next choice option on the path, this choice option is retrieved (line 6). If the number of this choice option is `0`, it is assumed that the `true` case of this choice point should be traversed. `true` is returned accordingly. If the condition in line 5 does not evaluate to true, there is no next choice option on the predetermined path. The third case (lines 8–18) deals with this more complex case. A new choice is created (line 8). The first choice option (for which `ChoiceOption.NUMBER` is `0`) of this choice contains the constraint c as the choice option's constraint. The second choice option contains the negation of c. The

[15] It is also possible to create choice points with more than two alternatives. This, however, is currently not done for simplicity.

current execution's instance of `SymbolicExecution` is asked to check which choice option of the new choice should be evaluated (line 9). This decision is delegated to the `MulibExecutor`'s search strategy (more details on this are given in Subsection 6.6). It is important to remark that only in this third case a new constraint can be pushed to the `SolverManager`, namely the constraint of the `ChoiceOption` that is selected this way. It can also occur that no choice is evaluated next (lines 10–12). In this case, we throw the specialized runtime exception `Backtrack` (line 12). Since we do not restore the runtime's state by means of a trail, but rather reinvoke the search region method, the term *backtracking* here is used rather loosely to define that we stop exploring a certain choice option in the search tree. `Backtrack` is a singleton that does not collect any information on the stack trace. Hence, throwing this exception is a relatively performant and simple way to terminate the search region at any position in the code. Different to `Fail`, `Backtrack` does not mark the choice option as unsatisfiable. Instead, both choice options are added to the choice options that must be evaluated (line 11). This might also trigger the start of parallel `MulibExecutor`s managed by the same `MulibExecutorManager` (not depicted in the Algorithm). The parallel `MulibExecutor`s will start their evaluation loop (recall Figure 4) by evaluating one of the returned choice options. Otherwise, only the choice option that is not evaluated is added to the choice options that must be evaluated later (lines 14–17). Then, it is checked whether the number of the chosen choice option, $coNumber$ is equal to `0` (line 18). If this is the case, the `true`-branch should be evaluated. Otherwise, the `false`-branch is evaluated.

Note that, since every occurrence of `Throwable` in the search region is substituted by an instance of `PartnerClassThrowable` (recall Subsection 4.2) neither `Backtrack` nor `Fail` are ever caught in the transformed program's code since they are not subclasses of `PartnerClassThrowable` and are not transformed. If, in a program, `Throwable` is caught, in the transformed program, `PartnerClassThrowable` is caught instead.

As `ChoicePointFactories` are shared by all `MulibExecutor`s for this execution, they must be thread-safe.

Mulib also comes with another implementation of `ChoicePointFactory`, namely the `ConcolicChoicePointFactory`. This option is explained in more detail in Subsection 8.1.

6.5 Symbolic Values and Calculations

Similar to how (potential) choice points are treated during the program transformation, if indicator methods such as `Mulib.freeInt()` are used in the search region, or if calculations involve a symbolic value, bytecode instructions are substituted by appropriate method calls to the Mulib search framework.

For instance, `Mulib.freeInt()` is substituted by a call to `SymbolicExecution.symSint()`. `SymbolicExecution` does not implement the business logic to initialize those values but delegates this task to a `ValueFactory`. The default implementation is the `SymbolicValueFactory`. In this factory, if, e.g., a `SymSintLeaf` should be created, we first check whether we have already created and cached a `SymSintLeaf`. We do so by checking the current value of `SymbolicExecution.nextNumberSymSintLeaf`. In other words: `SymbolicValueFactory` caches created leaves and reuses them in every execution. Non-leaf `SymSint`s that wrap a DAG represented by an `Expression` (recall Subsection 4.1) are not cached in this implementation. The standard implementation also optionally enforces global domain constraints for primitive data types (see Subsubsection 8.2.5) and is thread-safe. More complex objects, such as instances of `PartnerClassObject` too are initialized by a `ValueFactory`. In this factory more tasks, such as restricting the length of `Sarray`s and the identifier of any `PartnerClass` are taken care of. For more details on this, the reader is referred to Winkelmann and Kuchen [53].

On the other hand, if a bytecode instruction, such as $iadd$ [33], potentially involves a symbolic value or if an indicator method call is used, the instruction is substituted by a method call such

as `Sint.add(Sint, SymbolicExecution)` where `SymbolicExecution` is used to delegate the call to an instance of `CalculationFactory`. In the `CalculationFactory`, it is checked whether such operations can proceed concretely, or whether a DAG in the form of an `Expression` or `Constraint` (see Subsection 4.1) should be created. In any case, the output of any method call to a `CalculationFactory` typically is created by a `ValueFactory`. The `CalculationFactory` also is used for representing objects for the constraint solver. For a discussion on this, the reader again is referred to [53].

The factory pattern has been chosen to allow for an easier comparison of various caching approaches and even employing different evaluation strategies. As an example for this: Another implementation of `ValueFactory` is the `ConcolicValueFactory` and for the `CalculationFactory` the `ConcolicCalculationFactory` is provided, both of which are explained in Subsection 8.1. All factories again are shared among all `MulibExecutor`s.

6.6 Search Strategies and Budgets

As aforementioned, `MulibExecutor`s are responsible for specifying search strategies. Search strategies are the means to decide on which of the unevaluated choice options a new execution should be started with and whether or not a symbolic execution should backtrack or continue the execution at a choice point. The important classes for this choice option-selection mechanism are depicted in Figure 10. Since `MulibExecutor`s are also responsible for some additional coordination tasks such as synchronizing the `SolverManager` with the `SymbolicExecution`, copying input arguments and setting up static fields (see Subsection 6.2), `AbstractMulibExecutor` is an implementation of the template design pattern [27] and pre-implements these additional tasks. Subclasses of `AbstractMulibExecutor` must only implement two methods. The first one, `AbstractMulibExecutor.selectNextChoiceOption(ChoiceOptionDeque)`, decides on which choice option should be navigated to when starting a new execution of the search region (see Subsection 6.2). The second one, `AbstractMulibExecutor.shouldContinueExecution()` relates to the decision on whether the execution should proceed with one of the choice options when encountering a new choice. In the other case, `Backtrack` is thrown (recall Subsection 6.4).

Mulib comes with five pre-implemented search strategies: BFS, DFS, Iterative Deepening Depth-First Search (IDDFS), Iterative Deepening Deepest Shared Ancestor Search (IDDSAS), and Deepest Shared Ancestor Search (DSAS). To implement a search strategy, subclasses of `MulibExecutor` can either rely on directly accessing the `ChoiceOptionDeque` or by navigating the search tree using `ChoiceOption`s (see Subsection 6.1). If a choice was encountered, those choice options that are not chosen to continue the execution with (lines 11 and 17 in Algorithm 5) are added to a double-ended queue. `ChoiceOptionDeque` is an ordered deque that orders `ChoiceOption`s according to their depth in the search tree.

BFS is implemented by overriding `selectNextChoiceOption(ChoiceOptionDeque)` to simply invoke `ChoiceOptionDeque.pollFirst()`. In turn, `shouldContinueExecution()` always returns `false`. DFS is implemented by invoking `ChoiceOptionDeque.pollLast()` and always deciding to continue the execution. IDDFS is implemented by invoking `ChoiceOptionDeque.pollFirst()` and using a runtime budget (see Subsubsection 8.2.3) to determine whether the execution should continue: Each time a new choice is uncovered, a counter is incremented. If the counter exceeds the specified budget, `Backtrack` is thrown (recall Algorithm 5, lines 10–12). In consequence, setting this budget to `1` degenerates IDDFS to BFS in terms of its search behavior.

A more complex search algorithm is IDDSAS. To start an execution with IDDSAS, we take the last evaluated choice option into account and try to find an unevaluated choice option that shares a deepest ancestor choice option. For this, the choice options of the search tree are traversed (see Subsection 8). This is equivalent to the DSAS procedure described in earlier work [52]. Since neither the head nor the tail of the deque is polled directly, `ChoiceOptionDeque.request(ChoiceOption)` is used to remove the choice option from the deque. If the choice option has already been re-

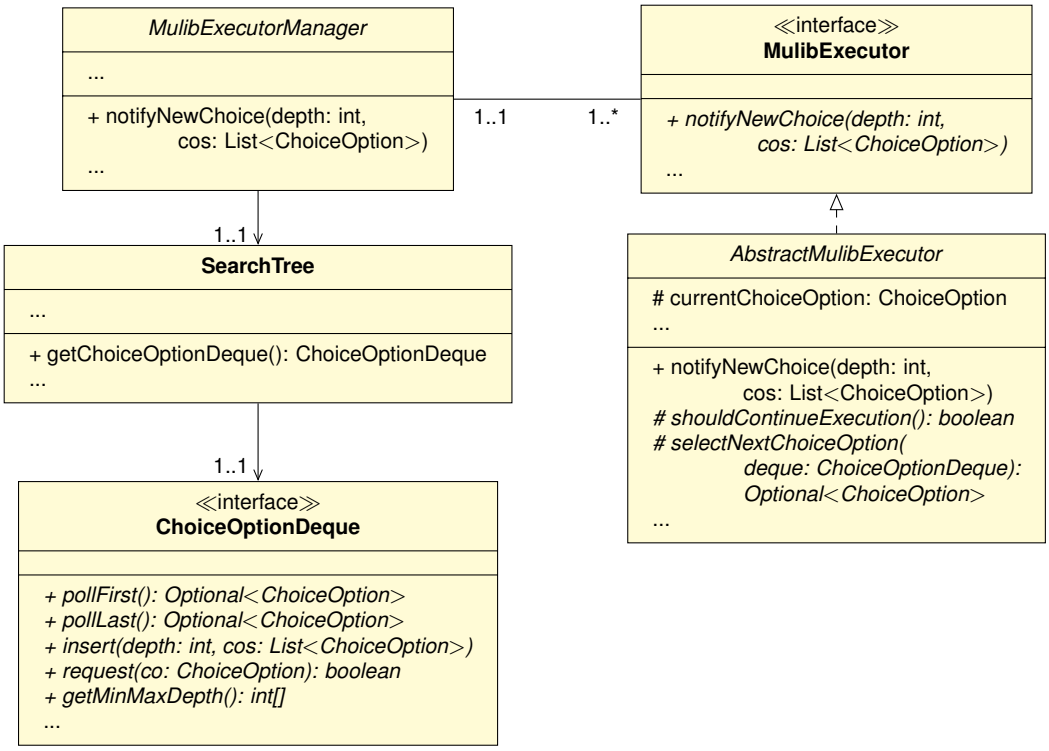

Figure 10: The fundamental classes for performing search in Mulib.

moved by another executor, i.e., another executor is already evaluating this choice option, the tree is ascended to search for another unevaluated choice option. On the other hand, for determining whether or not we should continue with the execution of the search region, the minimum and maximum depth of the choice options in the deque is computed via `ChoiceOptionDeque.getMinMaxDepth()`. The aim is that, for as long as the minimal depth is not equal to the maximal depth, the current execution should only be able to explore choice options the depth of which is limited by the maximal depth. Thereafter, it is allowed to increase the maximal depth of a choice option via a fixed increment (see Subsubsection 8.2.3).

Table 3 offers an overview of the various search strategies, whether they are complete, and, based on the insights of earlier work [52], their expected efficiency to evaluate a finite search region when combined with an incremental constraint solver. This expected efficiency is determined qualitatively based on different degrees to which the search strategies make use of incremental constraint solving (see Winkelmann and Kuchen [52]). In practice, the budgets, chosen constraint solver, structure of the search tree and types of constraints are factors that must be regarded.

The term *complete* here refers to the ability of guaranteeing to return a solution if one exists. However, a noteworthy exclusion of scope are loops and recursion calls that do not terminate without involving a symbolic value. For instance, given a loop such as `while(true){...}`, if the body does not include a reachable `break` statement, a complete search algorithm, as it is defined here, still is not guaranteed to find all reachable solutions.

When using a single `MulibExecutor`, DFS is expected to perform best (with a minimal edge over DSAS due to constant overhead). DSAS is preferable if multiple `MulibExecutor`s are employed in parallel [52]. If a complete search strategy is desirable, IDDSAS should be preferred over BFS and IDDFS. However, it should be noted that there are solver implementations that often can mitigate the loss of incremental performance due to non-locally chosen choice options (see Subsubsection

Name	Complete	Expected efficiency w.r.t. incremental constraint solvers	Expected efficiency-loss w.r.t. incremental constraint solvers in a parallel setting
DFS	No	+++	$--$
DSAS	No	+++	0
IDDSAS	Yes	++	0
IDDFS	Yes	+	0
BFS	Yes	0/+	0

Table 3: The completeness of the search strategies, as well as the expected efficiency (loss).

7.2.2). The interested reader is referred to earlier work [52] for a discussion of non-locally chosen choice options and their effect on performance. Also, it should be remarked that in the cases of both IDDFS and IDDSAS, their performance with incremental constraint solvers is dependent on the budget determining the size of increment. In fact, if choosing a large enough incremental budget for IDDSAS, it degenerates to DSAS.

6.7 Differences to Muli$_{sjvm}$

The search tree representations of Muli$_{sjvm}$ and Mulib differ only in details. The most noteworthy differences are the inclusion of an `ExceededBudget` node since Mulib can define several budgets (see Subsection 8.2) and the addition of a deque, simplifying the retrieval of unevaluated choice options using multiple search strategies in parallel.

Mulib's approach to use various factories, namely `ChoicePointFactory`, `ValueFactory`, and `CalculationFactory` facilitates the experimentation with different settings and caching and even allowed the implementation of concolic execution in Mulib.

The approaches to search of Muli$_{sjvm}$ and Mulib are fundamentally different. Muli$_{sjvm}$ utilizes forward- and backward-trails for resetting the program state to the state when first encountering a choice point. If a search algorithm determines that a certain choice option should be evaluated next, Muli$_{sjvm}$ navigates the search tree like Mulib. Yet, during this navigation, forward- and backward-trails are applied to restore the state of the program to the state when originally finding the navigated-to choice option. In consequence, the search tree of the old Muli implementation contains several trail elements for, e.g., reverting the mutation of an object's field, changing the frame because a new method was invoked, and changing the program counter of the logic virtual machine [21]. In summary, search in Muli$_{sjvm}$ is rather fine-grained and low-level, requiring knowledge on technical details of a JVM. Additionally, since Muli$_{sjvm}$ is implemented in Java, there is considerable overhead for applying the trails. For each (inverse) application of a trail element, several Java bytecode instructions are executed. Moreover, it is hard to adapt Muli$_{sjvm}$ to account for symbolic execution of the same search region using multiple threads: The trail elements are changing the state of the virtual machine. For instance, when changing a frame `vm.setCurrentFrame(...)` is called, changing the frame the SJVM should execute next [1]. This is hard to combine with a parallel evaluation of the search space. As another example, trail elements for setting static variables would change the static variable for all parallel executors.

In contrast, the search algorithms of Mulib are defined on a higher level of abstraction. The search tree is navigated, yet, instead of applying trails, the search region is executed from the start each time with a predetermined sequence of choice options to follow. Beforehand, Mulib copies input arguments and, during the runtime, static variables. The first advantage is that Mulib does not have a nested JVM and can use a high-performance JVM. Judging by previous

experiments [52], repeating the same calculations redundantly has lower overhead than applying a fine-grained trail in many scenarios. The most relevant part from a perspective of performance, the incremental constraint solvers, still can be used accounting for backtracking. Moreover, Mulib's approach of solely storing a predetermined path of search options through the search tree is more memory efficient to the point where the performance of Muli$_{sjvm}$ can degrade over time or an `OutOfMemoryError` is thrown, while Mulib produces consistent run time measurements [52]. The higher level of abstraction also allows for rather concise implementations of search strategies: The search strategy-specific implementation of all five search strategies takes less than 100 lines of code in total.[16]

Finally, Mulib allows to specify various search strategies for which `MulibExecutor`s are generated that disjointly evaluate the search region in parallel. This includes the addition of two search strategies that perform well with incremental constraint solvers even if there are multiple competing executors [52].

Importantly, while Mulib currently does not implement more involved procedures for copying input arguments and static variables, such as copy-on-write data structures [20], this option is not excluded in the future. The fact that little thought has been given to this aspect of execution is mainly due to its, for the used set of benchmark examples, negligible effect on overall performance. Similarly, in the future, Mulib might implement a program transformation for not executing the search region from the start for every execution. Adaptations of the continuation-passing style, known from functional programming [2], might be a noteworthy idea, where continuations could be stored in the form of lambdas in `ChoiceOption` objects. However, due to the more complicated runtime environment and the small expected performance gain (if any) for the current set of examples for Mulib, such a setting should be optional. After all, most complexity stems from solver calls to evaluate the satisfiability of a constraint stack or for creating a model for labeling.

[16] They are implemented in the class `de.wwu.mulib.search.executors.GenericExecutor` and `de.wwu.mulib.search.executors.GlobalIddfsSynchronizer`. The modification of using a control-flow graph to dedicatedly search for choice options representing uncovered edges in the control-flow graph is not counted.

7 Solving

It is a well-known fact that for most programs, symbolic execution will spent most time on performing solver-calls and creating models from the solver for labeling symbolic values [11]. In consequence, it is desirable to make it easy for developers to add new solvers to the symbolic execution engine, thus allowing to compare results, and make solver calls as efficient as possible. In Mulib, new solvers can be added by means of an adapter. Said adapter should implement the `SolverManager` interface. In the following, first, the existing architecture of Mulib's adapted solvers is explained. Thereafter, the offered solvers are outlined in their characteristics. The abstract strategy for calculating multiple `Solution`s, opposed to `PathSolution`s (recall Subsection 3.3) is explained in Subsection 7.3. Finally, again, the differences between Muli$_{sjvm}$ and Mulib are discussed in Subsection 7.4.

7.1 Architecture

Similar to `AbstractMulibExecutor` (see Subsection 6.6), Mulib's `AbstractIncrementalEnabledSolverManager` implements the `SolverManager` interface and provides a template. Instead of implementing all methods of `SolverManager`, smaller methods with an easier-to-grasp scope must be implemented. Figure 11 depicts the main interface methods that are overridden by `AbstractIncrementalEnabledSolverManager`, as well the methods required to be implemented by the latter.

`AbstractIncrementalEnabledSolverManager` uses three template parameters to unify the functionality of various implementing subclasses: First, the parameter `B` represents the solver's specific way of representing constraints. For instance, in Z3, this is `BoolExpr`. Adding constraints to the constraint stack is performed by first transforming Mulib's representation of constraints, `Constraint`, into the format of the solver using `transformConstraint(...)`. Thereafter, this format is pushed onto the solver using `addSolverConstraintRepresentation(...)`. If the constraint is added due to reaching a choice option, before adding the constraint, `solverSpecificBacktrackingPoint()` is called. `M` is the parameter representing a model. A model is what is used to derive labels. To calculate a model, the constraint stack must be satisfiable, and thus, `calculateIsSatisfiable()` is called beforehand. The parameter `AR` represents the solver's specific representation of arrays, if there is any.

In summary, `AbstractIncrementalEnabledSolverManager` coordinates the functionality of the abstract methods in a manner so that:

1. The results of satisfiability calculations and calculated models are cached for as long as they are valid, i.e., for as long as no new constraint is added or a backtracking operation has been conducted.

2. The structure used to represent arrays is updated if necessary. Since, e.g., Z3's representation of arrays is immutable, the routines in `AbstractIncrementalEnabledSolverManager` make sure that to store a new value in the array's representation, a new array can be created which is used subsequently. The new array is equal to the old array in all positions except for the position in which a new value is stored [51]. This is done by overriding `createNewArrayRepresentationForStore(...)`.

3. The labeling-procedure is implemented on a high level of abstraction so that only the method `labelSymSprimitive(...)` must be overridden. The labeling-process takes symbolic aliasing, where multiple symbolic objects might represent the same concrete object [53], and remembered values [45] into account.

4. The procedure for retrieving multiple solutions from an initial solution is implemented on a high level of abstraction. More details are given in Subsection 7.3. Adapted solvers offering more efficient methods can override this method.

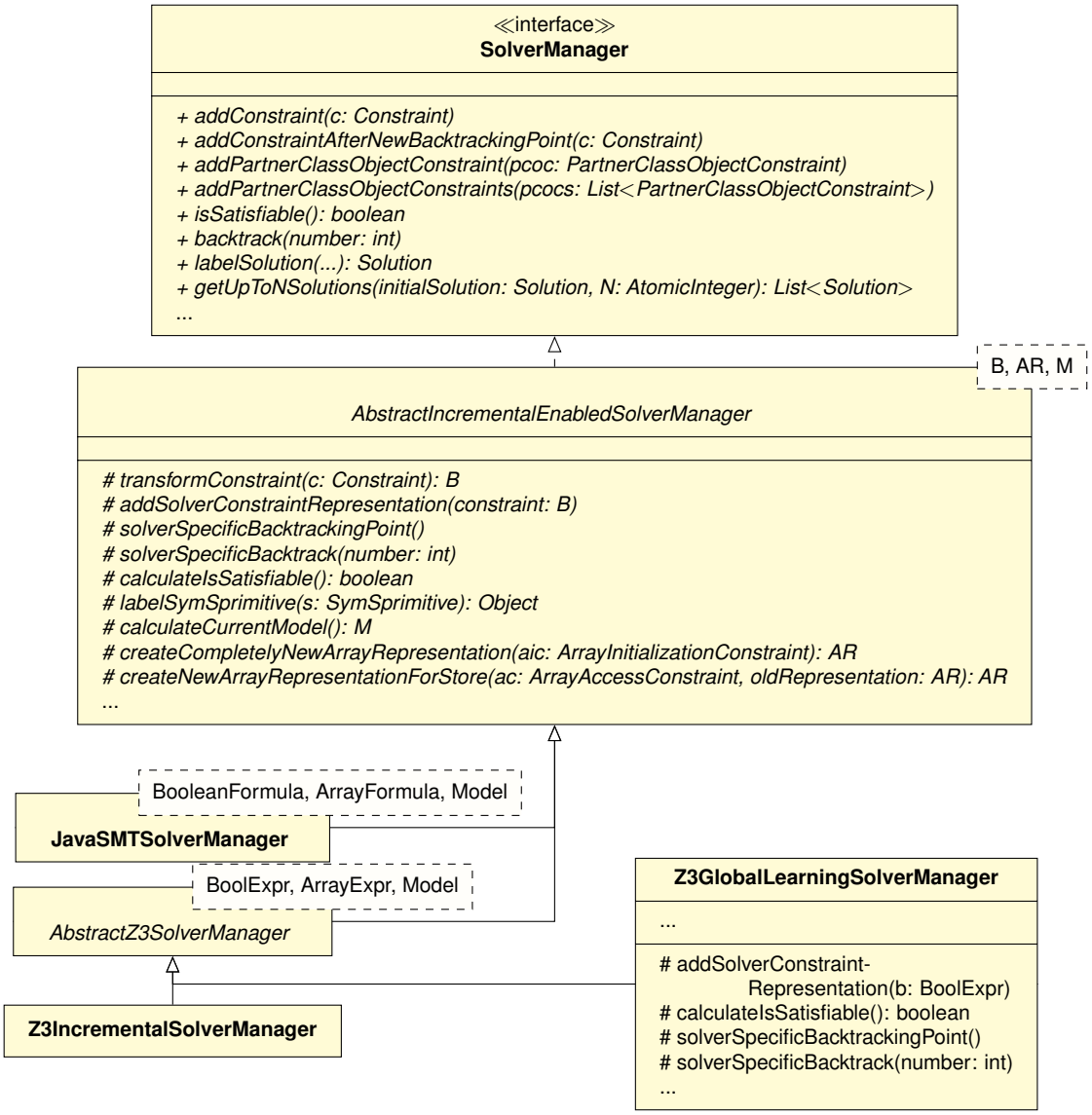

Figure 11: Simplified view on the fundamental classes for integrating new solvers into Mulib.

7.2 Offered Solvers

Mostly, during its development, Mulib has been used in combination with SMT solvers [5]. Most prominently, Z3 [24] has been used. Due to this reason, Mulib implements a dedicated Z3 adapter with two separate approaches. JavaSMT [3] is a separate adapter library that can be used to adapt several solvers using one common interface and that, in turn, is used by Mulib in an adapter.

7.2.1 Z3IncrementalSolverManager

The `Z3IncrementalSolverManager` is the standard solver manager for Mulib. In it, each backtracking point is represented by a scope. Due to these scopes, learning of lemmas for efficient solving is incremental [34]. In consequence, those search strategies that do not explore paths with a (large)

shared prefix of choice options only gain a small performance improvement from this learning. Examples for this are `BFS` and `IDDFS`, especially if the incremental budget is set to a low value. Both search strategies poll the choice options that are the highest in the search tree. Thus, the prefix of shared choice options is expected to be very small when compared to the last evaluated choice option of the previous execution. DFS performs well if no parallel execution is utilized, since otherwise, the different executors can ”steal” the choice options local to another executor (see Winkelmann and Kuchen [52] for more details). DSAS and IDDSAS choose their choice options from their respective neighborhood. The performance of IDDSAS is expected to improve if a larger incremental budget is used.

7.2.2 Z3GlobalLearningSolverManager

As aforementioned, some search strategies, such as BFS and IDDFS do not always benefit from incremental solving. However, in some instances, their particular procedure of search might be beneficial. For this reason, the `Z3GlobalLearningSolverManager` has been included. It implements a variant of a technique that is used to make learning possible without any local scopes [7]. This, however, is associated with an increased use of memory and some overhead. The `Z3GlobalLearningSolverManager` does not use local scopes. In consequence, `solverSpecificBacktrackingPoint` is a no-op. Instead, for each added constraint c_i, an additional `BoolExpr` a_i is constructed and a new constraint $a_i \rightarrow c_i$ is added. The antecedents a_i are maintained on a stack separate from the constraint solver and are added alike scopes/backtracking points, for each new choice option. If the satisfiability is to be calculated, the current stack of a_i is enforced to be true. In consequence, the solver has one single global scope and still can account for backtracking operations. All constraints $a_i \rightarrow c_i$ can be pushed on a single global scope of Z3. Those constraints will not be discarded and thus, learning of lemmas can occur in this global scope. For search algorithms that make good use of incremental solving, the incremental variant of Z3 usually still is more performant, as there is no overhead for propagating the antecedents for the implications. Furthermore, the incremental solver is more memory efficient, as there is no need to preserve all added constraints and the incremental scopes can be garbage collected.

7.2.3 JavaSMTSolverManager

Mulib additionally uses the external adapter library JavaSMT [3]. JavaSMT is an open source project for adapting, at the time of writing, nine SMT solvers, among which are:

1. Boolector [8],
2. CVC4 [6],
3. CVC5 [4],
4. MathSAT5 [13],
5. OptiMathSAT [40] (an extension of MathSAT5 for optimization),
6. Princess [39],
7. SMTInterpol [12],
8. Yices2 [25],
9. and Z3 [24].

With the implementation of `JavaSMTSolverManager` all solvers can be used with a simple change in Mulib's configuration (see Subsubsection 8.2.1). Particularly SMTInterpol is noteworthy as it is, like Mulib, written in Java and relatively performant. It should be noted that all SMT solvers have varying capabilities and strengths.

7.3 Retrieving Multiple Solutions From a Single Path

`AbstractIncrementalEnabledSolverManager` implements an approach to retrieve multiple solutions from a single path. This algorithm is called if the user, e.g., explores the search region using either `getUpToNSolutions(...)`, `getSolutionStream(...)`, or `getSolutionIterator(...)` (recall Subsection 3.6). Such a wrapper method is needed to extract additional solutions from a single path since SMT solvers typically do not implement a dedicated way to retrieve all solutions that are satisfiable for a set of constraints.[17] Note that the implemented way of `AbstractIncrementalEnabledSolverManager` can be overridden if the adapted solver offers a more efficient approach for enumerating solutions. Algorithm 6 depicts the standard implementation of `SolverManager.getUpToNSolutions(...)`.

The method is seeded with the `Solution` of a `PathSolution` (see Subsection 3.3) carrying the symbolic values and their labels (line 1). It is desired to retrieve alternative labels for the path represented by the `PathSolution`. It is important to note that the term 'labels' here strictly refers to labels of the return value and labels of remembered values. If the search region is called to return a `Stream` or `Iterator` of `Solution` objects (recall Subsection 3.3), the algorithm must account for being recalled for the same path. To achieve this, some local state in the `AbstractIncrementalEnabledSolverManager` is utilized that is explained at the end of this subsection. For now it is assumed that Algorithm 6 is called exactly once and `this.latestSolution` is null (line 2).

In a first step, the initial solution is stored in the field `this.latestSolution` (line 3). All `MulibExecutor`s and their respective `SolverManager`s are tasked with retrieving `N` solutions in total and so a shared instance of `AtomicInteger` is used to evaluate how many more solutions must be generated (lines 5 and 6). Next, the labels of the solution object are regarded (line 7). For each of the labels, we generate a *not-equals*-constraints (line 7). If, for instance, the label of a free `int i` is `42`, the constraint $i \neq 42$ is generated. For arrays, both symbolic indices and symbolic elements are regarded. For non-array objects, the value of each field is regarded. Symbolic aliasing is treated by generating a not-equals-constraint for the identifier, retrieved via `PartnerClass.__mulib__getId()`, and the identifier's current label (see Winkelmann and Kuchen [53]). This list of not-equals-constraints is then disjoined (line 8) and added as a new constraint (line 9). If the new constraint stack is satisfiable, this means that there is another labeling that satisfies the constraint stack before invoking `getUpToNSolutions(...)`. Thus, in this case, we label a new solution, add it, decrement `N`, and regard this new solution in the next iteration (lines 10–14). Therefore, in the next iteration, not-equals-constraints for this new solution are created and added to the constraint stack. If the new constraint stack was not found to be satisfiable the loop breaks and the field `this.latestSolution` is set to null (lines 15–17). Finally, the list of additional solutions for the existing path solution is returned (line 18).

[17] Traditionally, SMT solvers rather are concerned with whether a constraint stack is satisfiable at all and retrieving counter examples or examples [23].

Algorithm 6: Simplified procedure for retrieving multiple solutions from a single path.

```
1  getUpToNSolutions(Solution initialSolution, AtomicInteger N): List<Solution>
2      if this.latestSolution == null then
3          this.latestSolution = initialSolution;
4      List<Solution> result = new ArrayList<>();
5      int currentN = N.decrementAndGet();
6      while currentN > 0 do
7          List<Constraint> neqConstraints = getNeqConstraints(this.latestSolution.labels);
8          Constraint disjunction = Or.newInstance(neqConstraints);
9          addConstraint(disjunction);
10         if isSatisfiable() then
11             Solution newSolution = labelSolution(...);
12             result.add(newSolution);
13             currentN = N.decrementAndGet();
14             this.latestSolution = newSolution;
15         else
16             this.latestSolution = null;
17             break;
18     return result;
```

To allow for the on-demand retrieval of additional `Solution`s, the `AbstractIncrementalEnabeledSolverManager` maintains the latest solution retrieved via aforementioned algorithm. Consider the case where the user is not content with the initial solution and requests an additional solution. In this case, Algorithm 6 is called for the first time with N set to 1. This case might occur if the user employs the `SolutionIterator` (recall Subsection 3.3) and calls `Iterator.next()` a **second** time.[18] In this case, the algorithm behaves as described before. Now, consider the case where Algorithm 6 is called for a second time for the same path. In this case, the field `this.latestSolution` is not overwritten and does not match the parameter $initialSolution$ (lines 2 and 3). Thus, the latest solution from a previous invocation of Algorithm 6 is regarded instead and not-equals-constraints are created for it, instead of the initial solution (lines 7–9). Finally, if there are no more solutions on the path (lines 17–19), the solver manager resets the `this.latestSolution` field and awaits to be fed another initial `Solution` from another `PathSolution`.

7.4 Differences to Muli$_{sjvm}$

Due to the integration of JavaSMT and aforementioned implementation of Z3 that can speed up solving even for BFS and IDDFS, Mulib implements several more constraint solvers than Muli$_{sjvm}$. Muli$_{sjvm}$ implemented support for JaCoP [30] and an incremental version of Z3. Mulib facilitates the addition of new constraint solvers by making use of the template design pattern: Subclasses of `AbstractIncrementalEnabledSolverManager` solely have to implement a small set of methods to assure compliance with Mulib. Mulib furthermore extends the mentioned functionalities by providing support for a custom approach for dealing with symbolic arrays and objects (see Winkelmann and Kuchen [53]). While in Muli$_{sjvm}$ only Z3's custom array theory is supported for primitive-typed values, Mulib additionally allows to use an oftentimes more performant approach for dealing with array constraints that can also make use of symbolic aliasing to represent arrays of reference-typed elements [53].

However, currently Mulib does not implement support for free objects with a free type [22]. A discussion of this is provided in Subsection 9.3.

[18] Retrieving the first `Solution` does not require a call to Algorithm 6. Instead, the initial solution from the `PathSolution` is used.

8 Further Notes

Mulib can be configured extensively and even the type of execution can be adapted. In the following, first Mulib's concolic execution approach is described. Thereafter, several important configuration settings are explained.

8.1 Concolic Execution

Mulib offers both a purely symbolic execution, where the input values are unknown for the entire execution until finally labeling the values when extracting a `PathSolution`, and a concolic execution mode. In concolic execution both a concrete and a symbolic value state are carried through the program where the concrete value state is one possible labeling of the symbolic value state [11]. This can be beneficial because when encountering a new choice point, e.g., `if (i >= 42) {...}`. The concrete value state can be used to determine a choice option that must be satisfiable without needing to check the constraint solver. If the concrete value state for variable `i` states that `i` can be labeled to `1337`, it is known that the constraint $i >= 42$ must be satisfiable without having to contact the constraint solver. In consequence, this path can be taken.

To implement this, the Mulib framework type system (see Figure 6) was extended by two further types that are depicted in Figure 12.

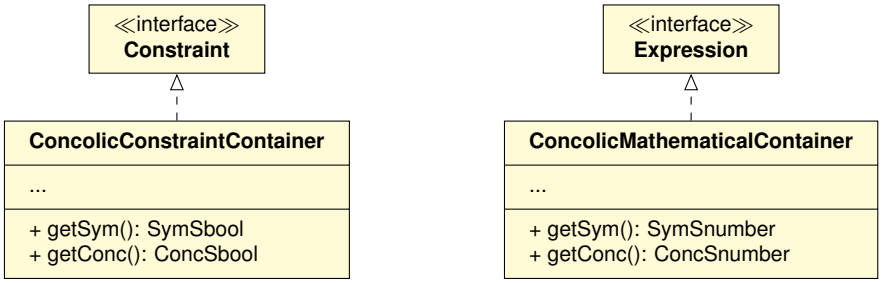

Figure 12: The two types that carry both the symbolic and the concrete value states.

If any new symbolic leaf value is created, immediately, the `SolverManager` is asked to label it. Both the symbolic value as well as its label then are wrapped in an instance of `ConcolicConstraintContainer`, if a `SymSbool` is created, or `ConcolicMathematicalContainer`, if any `SymSnumber` is created. The respective container is then in turn wrapped in an instance of `SymSbool` or a `SymSnumber`. This behavior is implemented using the `ConcolicValueFactory`. In turn, the `ConcolicCalculationFactory` performs executions both on the symbolic, as well as on the concrete representation. If two symbolic numbers are added, a `Sum` object is created for the symbolic values (retrieved via `getSym()`) while the labels (retrieved via `getConc()`) are added concretely. Finally, the `ConcolicChoicePointFactory` extends Algorithm 5 by pre-selecting the choice option that is already satisfied by the concrete labels. In consequence, when a new choice option is discovered, the constraint solver does not need to be called immediately since it is known that one of the choice options is satisfiable.

8.2 Configurability

Each execution of a search region, e.g. triggered via `Mulib.getPathSolutions(...)`, can be configured using a `MulibConfigBuilder` (see Section 3). In the following, the most important configuration settings are outlined.

8.2.1 Solver Configuration

1. SOLVER_GLOBAL_TYPE: Determines the solver manager an instance of which is used for each of the MulibExecutors.

2. SOLVER_HIGH_LEVEL_SYMBOLIC_OBJECT_APPROACH: Determines whether a solver-independent way of treating array constraints and symbolic aliasing should be used. This can be more efficient than built-in array theories. More details are given in Winkelmann and Kuchen [53].

8.2.2 Search Configuration

1. SEARCH_MAIN_STRATEGY: Determines the search strategy used by the MulibExecutor of the main thread.

2. SEARCH_ADDITIONAL_PARALLEL_STRATEGIES: Determines the number of other MulibExecutors that should be started. For each search strategy an own thread is spawned.

3. SEARCH_CHOICE_OPTION_DEQUE_TYPE: Determines the implementation for the choice option deque (see Subsection 6.6) that is used. There are two implementations: One, where a linked list is used. This is useful for single-threaded BFS and DFS. The other implementation offers a deque based on random-access so that also intermediate choice options, i.e., choice options that are not at the head or tail of the deque, can be retrieved more quickly.

4. SEARCH_CONCOLIC: Determines whether a pure symbolic or a concolic execution should be performed. Setting this to true will instantiate a concolic execution.

5. SEARCH_ALLOW_EXCEPTIONS: Determines whether, if an exception is found, the execution should be terminated or not. For instance, if the purpose of the symbolic execution is to find violated assertions, it can be sufficient to immediately indicate a failure and abort the execution.

8.2.3 Budgets Configuration

1. BUDGET_FIXED_ACTUAL_CP: Determines the maximum depth of the search tree. Choice options with a higher depth are not considered.

2. BUDGET_INCR_ACTUAL_CP: Determines the incremental budget used by IDDFS and IDDSAS (see Subsection 6.6).

3. BUDGET_GLOBAL_TIME_IN_NANOSECONDS: Determines after how much time the MulibExecutors should be stopped.

4. BUDGET_MAX_FAILS: Determines how many fails, thrown by Mulib.fail(), can at most be thrown before stopping the MulibExecutors.

5. BUDGET_MAX_PATH_SOLUTIONS: Determines how many path solutions can at most be extracted before stopping the MulibExecutors.

6. BUDGET_MAX_EXCEEDED_BUDGET: Determines how many choice points can at most be pruned away due to exceeding the maximum depth of the search tree (see 1) before stopping the MulibExecutors.

8.2.4 Array Behavior Configuration

1. ARRAYS_USE_EAGER_INDEXES_FOR_FREE_ARRAY_OBJECT_ELEMENTS: Determines whether, when accessing a free array of reference-typed values, a sequence of choice points should be constructed so that the index is immediately bound to one specific `int` value. If this is not true, the index remains symbolic and a symbolic object is returned.

2. ARRAYS_THROW_EXCEPTION_ON_OOB: Determines whether, for an access to an array where the length of the array or the index is symbolic, a choice point should be created. If the index can be larger than the length or less than zero, an `ArrayIndexOutOfBoundsException` is thrown. If this option is not set to `true`, no choice option is created and a new constraint is pushed to the constraint solver, assuming that the index is in a valid range.

8.2.5 Value Domain Configuration

1. VALS_SYMSINT_LB: Can be set to determine a lower bound for all `SymSint` values. This can be overridden by using `Mulib.freeInt(int, int)`.

2. VALS_SYMSINT_UB: Can be set to determine an upper bound for all `SymSint` values. This can be overridden by using `Mulib.freeInt(int, int)`.

3. Analogously, for each type of primitive number, there is a setting for a lower and upper bound.

4. VALS_TREAT_BOOLEANS_AS_INTS: Even though Java code does not allow the addition of `boolean`s and `int`s, this is possible on the bytecode level. However, it is more efficient to treat `boolean`s as atomic constraints, rather than as proxy expressions alike `b == 1`. Thus, this setting is false by default. If set to true, `boolean`s are treated as numbers.

8.2.6 Transformation Configuration

1. TRANSF_IGNORE_CLASSES and TRANSF_IGNORE_FROM_PACKAGES: Determines which classes to ignore during the program transformation. Either concrete classes or a whole package can be given. These classes will not receive a partner class. This setting should be used with extreme care to not break up class hierarchies. It is useful for integrating Mulib with other tools following the program instrumentation/transformation approach, such as Dacite [46].

2. TRANSF_REPLACE_METHOD_WITH_OTHER_METHOD: Users of Mulib can add a map of method-pairs. The first method should be replaced by the second method. In consequence, during the program transformation, if the first method is called, instead, the second method is called. This is useful for substituting native methods.

3. TRANSF_REPLACE_TO_BE_TRANSFORMED_CLASS_WITH_SPECIFIED_CLASS: Users of Mulib can add a map of class-pairs. The first class should be replaced by the second class. In consequence, during the program transformation, if the first class should be used, instead, the second class is used. This is useful for substituting native methods and providing model classes. For instance, a database driver might be modeled via a Java class so that the database becomes part of the symbolic execution (see, e.g., Winkelmann and Kuchen [54]).

4. TRANSF_GENERATE_CHOICE_POINTS_WITH_ID: Determines whether choice point methods, substituting conditional jumps (see Enumeration Items 27 and 26 in Subsubsection 5.4.2) should be called with an additional constant long value. For instance, if `Sint.eqChoice(SymbolicExecution)` is called, if this setting is activated, `Sint.eqChoice(SymbolicExecution, long)` is called instead. The argument to the `long` parameter is a constant value that uniquely identifies this choice point in the

control-flow graph. If this setting is activated, all if-instructions are tainted so that an identifier for all branches of the control-flow graph can be generated.

5. `CFG_USE_GUIDANCE_DURING_EXECUTION`: When evaluating a choice, it is checked whether both options of this choice, representing edges in the control-flow graph, have already been covered. If one has not been covered, it is evaluated next. For this setting to work, configuration option `TRANSF_GENERATE_CHOICE_POINTS_WITH_ID` must be set to true. This is because the identifier of this choice point is used to identify the node in the control-flow graph.

6. `CFG_TERMINATE_EARLY_ON_FULL_COVERAGE`: Determines whether, if all edges of the control-flow graph have been covered, search should terminate. For this setting to work, configuration option `TRANSF_GENERATE_CHOICE_POINTS_WITH_ID` must be set to true. This is because the identifier of this choice point is used to identify the node in the control-flow graph.

9 Future Work

Since Java is a rather feature-rich language, integrating some features with Muli are an ongoing effort; - Mulib in its current state can still be enhanced. Three particularly interesting features are summarized in the following.

9.1 Free Enums

Since Mulib currently builds upon bytecode as the chosen representation, aside from native methods, additional effort has to be applied only for those types that are treated as special cases by the JVM [33]. Consequently, the pool of classes that needs to be modelled specifically is rather small. However, similar to `java.lang.Throwable`, there is another class that serves as an abstract supertype of other Java classes and that is treated as a special case. `java.lang.Enum` represents a class with a fixed number of instances [33, 28]. To treat it, a class `PartnerClassEnum` could be introduced (see Figure 6) which is enriched by the (transformed) fields of `Enum`. Initializing a variable of an enum `E` via `Mulib.freeObject(E.class)` must then initialize a symbolic alias of one of the instances defined in `E`.

9.2 Free Strings

At the moment of writing, Mulib does not support treating `String`s as symbolic values. `String`s are rather complex data types deserving custom treatment. Respective constraints are supported by several SMT constraint solvers [5] and do not solely treat the equality of `String`s, but also whether a symbolic string contains a substring or even matches a regular expression [56].

Integrating free strings into Mulib implies creating a new subtype of constraints, namely `StringConstraint`s for which the various subtypes, such as a constraint for regular expressions, should be created. It also implies creating a new library class `Sstring`, extending `PartnerClassObject` and providing the methods of `java.lang.String`. The static methods of `java.util.regex.Pattern` and `java.util.regex.Matcher` also should be regarded.

9.3 Free Objects

Mulib currently does not implement support for free objects [22]. The reason for this is that the new implementation should also offer an option to delegate the differentiation of which type the object has to the constraint solver. As an example: In Muli$_{sjvm}$, if an object o has a free type T, and some method $o.m()$ is called, for which subtypes of T have other implementations, Muli$_{sjvm}$ would create a new choice. The number of choice options would be the number of these implementations provided by at least one non-abstract class [22].

Yet, in the work on enabling the support for free arrays of reference-typed elements, a clever delegation to the constraint solver was found to be vastly more efficient in many cases [53]. For instance, consider the following method:

```
A getA(A[] as, int i) {
  return as[i];
}
```

Consider the scenario where `getA(...)` is called with an array `as` with a high length and a symbolic index `i`. In this scenario, Mulib does not need to spawn different values for `i` and execute the program with each one. Instead, a symbolic alias is spawned that refers to any object in `as` [53]. Determining which object is referred to is delegated to the constraint solver.

An analogous delegation cannot occur with Muli$_{sjvm}$'s approach for free objects: In Muli$_{sjvm}$'s implementation, constraints involving the type of a object are treated completely in separation from the main constraint solver. Instead of representing the object type as a value in the constraint solver, Muli$_{sjvm}$ uses a separate structure [41] to compute the allowed types via set operations. This is problematic when trying to account for symbolic aliasing: Rather than delegating the distinction to the constraint solver, it is required that a number of choice options is spawned to account for typing. These choice options could be represented in a constraint instead. In the future, Mulib should strive to integrate constraints involving types of free objects into a solver. Note that this implies including constraints that are created while invoking a method on an object with a free type. Instead of spawning a choice option for each possible implementation invoked by this object [22], the method invocations should be summarized without creating a choice point. This is a rather non-trivial problem. Furthermore, the efficiency of this summarization is context dependent. Hence, a future implementation should offer a configuration option activating and deactivating this behavior.
The second issue with free objects is that using Mulib's indicator methods do not allow for specifying type variables. Type variables, among class types, interface types and array types, make up the four kinds of reference types in Java [28]. In contrast, the JVM specification only defines three reference types [33]. Given a type variable `T`, it is not valid to access `T.class`. This is because the Java compiler applies type erasure, where the left-most bound is utilized instead [28]. A left-most bound of a type variable `T` which was restricted via `<T extends U & V & ...>` is `U`. Since `U` does not capture all type information of `T` in general, `T.class` is not allowed. For a statement such as `T t free;` Muli$_{sjvm}$ assumes that the left-most bound should be initialized as a free value, i.e., `U t free;` is assumed. The fact that the intersection with, e.g., `V` is required is not accounted for. Earlier work on Muli thus left type variables to future work [22]. The most forward approach to treat the semantics of type intersections in Mulib is to split the initialization into a two-step procedure: Aforementioned free object declaration `T t free;` could be mapped to a sequence of `T t = Mulib.freeObject(U.class);` followed by `Mulib.assume(t instanceof V && t instanceof ...);`. A transpiler, as mentioned in Subsection 3.1 could automatically generate the two steps for a `free` declaration involving type variables.

9.4 Folding Instructions Initializing Potentially Symbolic Booleans

The example from Subsubsection 4.1.2 demonstrated how and (`&&`) and or (`||`) operations can be calculated in Java:

```
boolean and(boolean b0, boolean
    b1) {
    return b0 && b1;
}
```

```
boolean or(boolean b0, boolean
    b1) {
    return b0 || b1;
}
```

Instead of implementing specific bytecode instruction for such operations, Java instead compiles to and executes conditional jumps pushing either `0` (representing `false`) or `1` (representing `true`) onto the stack:

```
1     L0
2      ILOAD 0
3      IFEQ L1
4      ILOAD 1
5      IFEQ L1
6      ICONST_1
7      GOTO L2
8     L1
9      ICONST_0
10    L2
11     IRETURN
```

```
1     L0
2      ILOAD 0
3      IFNE L1
4      ILOAD 1
5      IFEQ L2
6     L1
7      ICONST_1
8      GOTO L3
9     L2
10     ICONST_0
11    L3
12     IRETURN
```

This is inefficient for symbolic execution since for each jump involving a symbolic `Sbool`, a new choice point is created. In the given example, it is not even necessary to perform a single jump; - instead, a single symbolic expression might be created. This becomes clear when regarding the (decompiled) transformed program that, applying the transformation described in Section 5, looks along the lines of:

```
1  Sbool and(Sbool b0, Sbool b1) {
2    SymbolicExecution se =
3        SymbolicExecution.get();
4    Sbool r;
5    if(b0.negatedBoolChoice(se)
6    || b1.negatedBoolChoice(se)){
7      r = Sbool.concSbool(false);
8    } else {
9      r = Sbool.concSbool(true);
10   }
11   return r;
12 }
```

```
1  Sbool or(Sbool b0, Sbool b1) {
2    SymbolicExecution se =
3        SymbolicExecution.get();
4    Sbool r;
5    if(b0.boolChoice(se)
6    && b1.boolChoice(se)){
7      r = Sbool.concSbool(true);
8    } else {
9      r = Sbool.concSbool(false);
10   }
11   return r;
12 }
```

In this transformed code the jumps (lines 4 and 5 respectively) are directly used to create a wrapper for the pushed `0`, i.e., `false` or `1`, i.e., `true`.

A better program transformation might create the following (decompiled) code:

```
1  Sbool and(Sbool b0, Sbool b1) {
2    SymbolicExecution se =
3        SymbolicExecution.get();
4    return b0.and(b1, se);
5  }
```

```
1  Sbool or(Sbool b0, Sbool b1) {
2    SymbolicExecution se =
3        SymbolicExecution.get();
4    return b0.or(b1, se);
5  }
```

Here, a dedicated method call (`Sbool.and(...)` or `Sbool.or(...)`) is used respectively. The conditional jumps and initialization of booleans are *folded* to a method call. These methods are already implemented in the Mulib search framework.

While for the given example, transformation approaches might be relatively straight-forward, in general, several things must be considered. For one, the `boolean` operations in Java are lazy. As such, any `boolean` value that is retrieved while potentially causing side-effects must be accounted for, i.e., the side-effect should still be caused in a lazy fashion. Furthermore, there are multiple ways of representing and and or operations using conditional jumps. All of these patterns should be accounted for. It might thus be easiest to, in a first step, exclude the `boolean` results of methods that are potentially lazily evaluated from this merging approach. Merging approaches might use existing functionalities for control-flow graphs, such as the ones offered by Soot.

Alternatively, one might focus on the eager variants to calculate and and or:

```
boolean and(boolean b0, boolean
    b1) {
    return b0 & b1;
}
```

```
boolean or(boolean b0, boolean
    b1) {
    return b0 | b1;
}
```

This will compile to:

```
1    L0
2     ILOAD 1
3     ILOAD 2
4     IAND
5     IRETURN
```

```
1    L0
2     ILOAD 1
3     ILOAD 2
4     IOR
5     IRETURN
```

Here, no choice points are created. This approach, however, still has several downsides: First, it is unintuitive to programmers that commonly use `&&` and `||` for boolean values, opposed to `&` and `|`. Second, extending the first point, when validating programs, more often than not, `&&` and `||` will be encountered. It would be beneficial if such a folding could be conducted for the use-case of verification and test case generation as well. Third, when combining multiple constraints using and and or, even when using `&` and `|`, choice points are still being generated. For instance, when checking whether a value is less than another value, the bytecode instruction if_lt is executed. This instruction again performs a conditional jump to initialize a boolean. Thus, in any case, some form of folding should be implemented, here creating, e.g., a call to `Sint.lt(Sint, SymbolicExecution)`.

Particularly in combination with the `Mulib.assume(Sbool)` method, such a folding is rather powerful. Consider the NQueens example from Subsection 3.3: If the bytecode is *folded* according to aforementioned meaning, not a single choice point has to be created and only a single satisfiability check must be conducted.

Even though this feature further complicates the transformation, it is worthwhile in terms of performance. Consider Table 4 displaying two constraint solvers integrated with Mulib solving the NQueens problem for 32 queens. Mulib was configured to employ either Z3, used as an incremental constraint solver, or CP-SAT. All measurements are given in seconds. Additionally, the minimum and maximum run time is specified. The measurements were taken after discarding five initial iterations to warm up the JVM. `NQueens-Incremental` implements the scenario where choice points are created for each added constraint. `NQueens-Folded` is Mulib framework-code that was implemented by hand to evaluate the impact of *bytecode folding*. In other words: `NQueens-Incremental` denotes a version where the runtime generates a choice point for each constraint. `NQueens-Folded` denotes a version where no additional choice options are added and only one constraint check is evaluated.

The chosen constraint solvers are Z3 and CP-SAT [14][19] Noticeably, CP-SAT and Z3 both perform better without creating any unnecessary choice options. Particularly, CP-SAT performs significantly better. The reason for this is that mostly SMT solvers, such as Z3, are dedicatedly designed for incremental constraint solving. In contrast, CP-SAT does not employ direct support for this.

This performance improvement again stresses the redesign of the mechanism employed for free objects so that no additional paths need to be spawned (see Subsection 9.3).

Additionally, aforementioned folding reduces the overhead of the Mulib runtime even more since

[19] CP-SAT has only been prototypically integrated with Mulib for now. The implementation of the respective adapter can be found under `https://github.com/NoItAll/mulib/blob/de661af81a4844e3c106a2d0037d96b317980d74/src/main/java/de/wwu/mulib/solving/solvers/CpSatSolverManager.java`.

Solver	NQueens -Incremental	NQueens -Folded
Z3	6.80 $[6.72; 6.92]$	5.51 $[5.47; 5.54]$
CP-SAT	64.44 $[60.86; 75.70]$	0.18 $[0.14; 0.24]$

Table 4: The mean run time of 15 iterations of solving the 32-Queens problem using Mulib.

Configuration	NQueens
Leftmost, Up, Step (default)	Timed out
FF, Up, Step	0.02 $[0.02; 0.03]$
FFC, Up, Step	0.04 $[0.02; 0.05]$
Min, Up, Step	Timed out
Max, Up, Step	Timed out

Table 5: The mean run time of 15 iterations of solving the 32-Queens problem using CLP(FD).

less backtracking must be conducted, Muli can get rather close to established programming languages, such as SWI-Prolog [50]. The NQueens problem, when modelled in Prolog using the CLP(FD) library [44], produces the measurements displayed in Table 5. All measurements are given in seconds. The implementation was retrieved from the official list of examples [42]. The configuration column describes the settings used for labeling [43].

Noticeably, the configuration for SWI-prolog, here given via the specification of the labeling strategy, can have an immense impact on the overall performance. In some instances, not even a single solution could be calculated after 5 minutes.

Note that the Mulib runtime has a benign impact on the overall execution duration, as is indicated in Table 6. In the table, the percentages of run time for solving the 32-Queens problem using either Z3 or CP-SAT are depicted for various aspects of the problem solving procedure. First, the column *Satisfiability & Model* comprises the percentage of the overall run time for calculating the satisfiability and for deriving suitable labels. The second column represents the percentages for the transformation from constraints of the Mulib type system to the solver-specific representation of constraints. The column *Initialization* represents the time fraction needed to initialize an instance of the respective solver. The used times result from solving one single instance of the problem. In consequence, the program transformation has to occur one initial time. This is depicted in the column *Program Transformation*.[20] The overall percentage of the runtime due to the constraint solver adds up to around 90% in the case of Z3 and 74% for CP-SAT. Additionally, one might argue that this percentage is increased for subsequent iterations where the program transformation does not need to be conducted.

Aside from bytecode folding, Mulib might research measures to tune solvers during execution.

[20] Note that this transformation has to occur only one single time for the first call to the search region.

Solver	Satisfiability & Model	Constraint Generation	Initialization	Program Transformation
Z3	89.20%	<1%	<1%	8.18%
CP-SAT	58.40%	8.37%	7.28%	13.67%

Table 6: The percentages of the overall run time for the 32-Queens problem.

10 Conclusion

This technical report describes several details, such as the overall workflow, the algorithms used for the program transformation, the procedure towards search, the integration of constraint solver, configuration options, as well as future research.

Mulib is a symbolic execution engine that employs the rather novel approach of program transformation (compare [31]). Search regions can be compiled to ordinary Java bytecode. The program transformation then generates partner classes that utilize the functionalities of the Mulib search framework. The Mulib search framework allows for a purely symbolic, i.e., non-concolic, execution. In consequence, Mulib does not rely on a custom SJVM. The search framework furthermore allows for different (parallely executed) search strategies and facilitates the integration of new constraint solvers. It was shown that, compared to Muli$_{sjvm}$, Mulib performs better in terms of run time and memory consumption [52]. Moreover, currently, Mulib can be executed using Java 17. Competing pure symbolic execution engines are often restricted to Java 8.

Furthermore, by means of Mulib, new concepts such as free arrays of reference-typed elements and ease-of-use features, such as `assume(...)` and `freeInt(0, 9)`, allowing for the direct specification of an upper and lower bound, could be introduced.

References

[1] *AbstractSearchAlgorithm*. Accessed: 2023-09-12. URL: https://github.com/wwu-pi/muli-env/blob/9df2800dcc34073442d50371c8532b1876d6e287/muli-runtime/src/main/java/de/wwu/muli/iteratorsearch/AbstractSearchAlgorithm.java#L253.

[2] A. W. Appel and T. Jim. “Continuation-Passing, Closure-Passing Style”. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, pp. 293–302. ISBN: 0897912942. DOI: 10.1145/75277.75303. URL: https://doi.org/10.1145/75277.75303.

[3] Daniel Baier, Dirk Beyer, and Karlheinz Friedberger. “JavaSMT 3: Interacting with SMT Solvers in Java”. In: *International Conference on Computer Aided Verification*. Springer. 2021, pp. 195–208.

[4] Haniel Barbosa et al. “CVC5: A Versatile and Industrial-Strength SMT solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 415–442.

[5] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. “The SMT-LIB Standard: Version 2.0”. In: *Proceedings of the 8th international workshop on satisfiability modulo theories (Edinburgh, UK)*. Vol. 13. 2010, p. 14.

[6] Clark Barrett et al. “CVC4”. In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer. 2011, pp. 171–177.

[7] Aaron Bembenek et al. “Datalog-based Systems Can Use Incremental SMT Solving”. In: *Proceedings of the 36th International Conference on Logic Programming*. 2020.

[8] Robert Brummayer and Armin Biere. “Boolector: An efficient SMT solver for bit-vectors and arrays”. In: *Tools and Algorithms for the Construction and Analysis of Systems: 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings 15*. Springer. 2009, pp. 174–177.

[9] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. “ASM: A code manipulation tool to implement adaptable systems”. In: *In Adaptable and extensible component systems*. 2002.

[10] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[11] Cristian Cadar and Koushik Sen. “Symbolic execution for software testing: three decades later”. In: *Communications of the ACM* 56.2 (2013), pp. 82–90.

[12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. “SMTInterpol: An Interpolating SMT Solver”. In: *International SPIN Workshop on Model Checking of Software*. Springer. 2012, pp. 248–254.

[13] Alessandro Cimatti et al. “The MathSAT5 SMT Solver”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 93–107.

[14] *CP-SAT Solver — OR Tools — Google for Developers*. Accessed: 2023-09-25. URL: https://developers.google.com/optimization/cp/cp_solver.

[15] T. Cramer et al. “Compiling Java just in time”. In: *IEEE Micro* 17.3 (1997), pp. 36–43. DOI: 10.1109/40.591653.

[16] Jan C Dageförde. “An Integrated Constraint-Logic and Object-Oriented Programming Language: The Münster Logic-Imperative Language.” PhD thesis. Westfälische Wilhelms-Universität Münster, 2020.

[17] Jan C Dageförde and Herbert Kuchen. “A Compiler and Virtual Machine for Constraint-Logic Object-Oriented Programming with Muli”. In: *Journal of Computer Languages* 53 (2019), pp. 63–78.

[18] Jan C Dageförde and Herbert Kuchen. “A Constraint-Logic Object-Oriented Language”. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. 2018, pp. 1185–1194.

[19] Jan C Dageförde and Herbert Kuchen. “Applications of Muli: Solving Practical Problems with Constraint-Logic Object-Oriented Programming”. In: *Analysis, Verification and Transformation for Declarative Programming and Intelligent Systems: Essays Dedicated to Manuel Hermenegildo on the Occasion of His 60th Birthday*. Springer, 2023, pp. 97–112.

[20] Jan C. Dageförde and Herbert Kuchen. “Retrieval of Individual Solutions from Encapsulated Search with a Potentially Infinite Search Space”. In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*. SAC ’19. Limassol, Cyprus: Association for Computing Machinery, 2019, pp. 1552–1561. ISBN: 9781450359337. DOI: 10.1145/3297280.3298912. URL: https://doi.org/10.1145/3297280.3298912.

[21] Jan C. Dageförde and Finn Teegen. “Structured Traversal of Search Trees in Constraint-Logic Object-Oriented Programming”. In: *Declarative Programming and Knowledge Management*. Ed. by Petra Hofstedt et al. Cham: Springer International Publishing, 2020, pp. 199–214. ISBN: 978-3-030-46714-2.

[22] Jan C. Dageförde, Hendrik Winkelmann, and Herbert Kuchen. “Free Objects in Constraint-Logic Object-Oriented Programming”. In: *23rd International Symposium on Principles and Practice of Declarative Programming*. PPDP 2021. Tallinn, Estonia: Association for Computing Machinery, 2021. ISBN: 9781450386890. DOI: 10.1145/3479394.3479409. URL: https://doi.org/10.1145/3479394.3479409.

[23] Leonardo De Moura and Nikolaj Bjørner. “Satisfiability Modulo Theories: Introduction and Applications”. In: *Commun. ACM* 54.9 (Sept. 2011), pp. 69–77. ISSN: 0001-0782. DOI: 10.1145/1995376.1995394. URL: https://doi.org/10.1145/1995376.1995394.

[24] Leonardo De Moura and Nikolaj Bjørner. “Z3: An efficient SMT Solver”. In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[25] Bruno Dutertre. “Yices 2.2”. In: *International Conference on Computer Aided Verification*. Springer. 2014, pp. 737–744.

[26] *FieldContainer*. Accessed: 2023-09-14. URL: https://github.com/wwu-pi/muggl/blob/85b53690bdbcd730fc200818357c7e580fd8bc0b/muggl-core/src/de/wwu/muggl/vm/initialization/FieldContainer.java#L34.

[27] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[28] James Gosling et al. *The Java® Language Specification - Java SE 17 Edition*. Aug. 2021. URL: https://docs.oracle.com/javase/specs/jls/se17/jls17.pdf (visited on 09/22/2023).

[29] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing”. In: *Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS’03. Warsaw, Poland: Springer-Verlag, 2003, pp. 553–568. ISBN: 3540008985.

[30] Krzysztof Kuchcinski. “Constraints-Driven Scheduling and Resource Assignment”. In: *ACM Trans. Des. Autom. Electron. Syst.* 8.3 (July 2003), pp. 355–383. ISSN: 1084-4309. DOI: 10.1145/785411.785416. URL: https://doi.org/10.1145/785411.785416.

[31] Henrich Lauko and Petr Ročkai. “LART: Compiled Abstract Execution: (Competition Contribution)”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2022, pp. 457–461.

[32] Henrich Lauko, Petr Ročkai, and Jiří Barnat. “Symbolic Computation via Program Transformation”. In: *Theoretical Aspects of Computing – ICTAC 2018*. Ed. by Bernd Fischer and Tarmo Uustalu. Cham: Springer International Publishing, 2018, pp. 313–332. ISBN: 978-3-030-02508-3.

[33] Tim Lindholm et al. *The Java® Virtual Machine Specification - Java SE 17 Edition*. Aug. 2021. URL: https://docs.oracle.com/javase/specs/jvms/se17/jvms17.pdf (visited on 09/19/2023).

[34] Tianhai Liu et al. “A Comparative Study of Incremental Constraint Solving Approaches in Symbolic Execution”. In: *Hardware and Software: Verification and Testing*. Ed. by Eran Yahav. Cham: Springer International Publishing, 2014, pp. 284–299. ISBN: 978-3-319-13338-6.

[35] Kasper Luckow et al. “JDart: A Dynamic Symbolic Analysis Framework”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Marsha Chechik and Jean-François Raskin. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, pp. 442–459. ISBN: 978-3-662-49674-9.

[36] Tim A. Majchrzak and Herbert Kuchen. *Muggl: The Muenster generator of glass-box test cases*. eng. ERCIS Working Paper 10. Münster, 2011. URL: http://hdl.handle.net/10419/58417.

[37] Malte Mues and Falk Howar. “GDart: An Ensemble of Tools for Dynamic Symbolic Execution on the Java Virtual Machine (Competition Contribution)”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 435–439. ISBN: 978-3-030-99527-0.

[38] Sebastian Poeplau and Aurélien Francillon. “Symbolic execution with SymCC: Don’t interpret, compile!” In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau.

[39] Philipp Rümmer. “A Constraint Sequent Calculus for First-Order Logic with Linear Integer Arithmetic”. In: *Proceedings, 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Vol. 5330. LNCS. Springer, 2008, pp. 274–289. ISBN: 978-3-540-89438-4.

[40] Roberto Sebastiani and Patrick Trentin. “OptiMathSAT: A Tool for Optimization Modulo Theories”. In: *International conference on computer aided verification*. Springer. 2015, pp. 447–454.

[41] *SolverManagerWithTypeConstraints*. Accessed: 2023-09-12. URL: https://github.com/wwu-pi/muggl/blob/53a2874cba2b193ec99d2aea8a454a88481656c7/muggl-solvers/src/de/wwu/muggl/solvers/SolverManagerWithTypeConstraints.java#L41-L65.

[42] *SWI-Prolog – Examples: Eight queens puzzle*. Accessed: 2023-09-25. URL: https://www.swi-prolog.org/pldoc/man?section=clpfd-n-queens.

[43] *SWI-Prolog – labeling/2*. Accessed: 2023-09-25. URL: https://www.swi-prolog.org/pldoc/man?predicate=labeling/2.

[44] Markus Triska. “The Finite Domain Constraint Solver of SWI-Prolog”. In: *FLOPS*. Vol. 7294. LNCS. 2012, pp. 307–316.

[45] Laura Troost, Hendrik Winkelmann, and Herbert Kuchen. “An Integrated Visualization Approach Combining Dynamic Data-Flow Analysis with Symbolic Execution”. In: *Proceedings of the 19th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. (accepted). INSTICC. SciTePress, 2024.

[46] Laura Troost. and Herbert Kuchen. “A Comprehensive Dynamic Data Flow Analysis of Object-Oriented Programs”. In: *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC. SciTePress, 2022, pp. 267–274. ISBN: 978-989-758-568-5. DOI: 10.5220/0010984800003176.

[47] Raja Vallée-Rai et al. “Soot: A Java Bytecode Optimization Framework”. In: *CASCON First Decade High Impact Papers*. CASCON ’10. Toronto, Ontario, Canada: IBM Corp., 2010, pp. 214–224. DOI: 10.1145/1925805.1925818. URL: https://doi.org/10.1145/1925805.1925818.

[48] Willem Visser and Jaco Geldenhuys. “COASTAL: Combining Concolic and Fuzzing for Java (Competition Contribution)”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2020, pp. 373–377.

[49] *VM*. Accessed: 2023-09-12. URL: https://github.com/DeepseaPlatform/coastal/blob/master/src/main/java/za/ac/sun/cs/coastal/symbolic/VM.java#L1405-L1427.

[50] Jan Wielemaker et al. “Swi-prolog”. In: *Theory and Practice of Logic Programming* 12.1-2 (2012), pp. 67–96.

[51] Hendrik Winkelmann, Jan C Dageförde, and Herbert Kuchen. “Constraint-Logic Object-Oriented Programming with Free Arrays”. In: *International Workshop on Functional and Constraint Logic Programming*. Springer. 2020, pp. 129–144.

[52] Hendrik Winkelmann and Herbert Kuchen. “Constraint-Logic Object-Oriented Programming on the Java Virtual Machine”. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC ’22. Virtual Event: Association for Computing Machinery, 2022, pp. 1258–1267. ISBN: 9781450387132. DOI: 10.1145/3477314.3507058. URL: https://doi.org/10.1145/3477314.3507058.

[53] Hendrik Winkelmann and Herbert Kuchen. “Constraint-Logic Object-Oriented Programming with Free Arrays of Reference-Typed Elements via Symbolic Aliasing”. In: *Proceedings of the 18th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*. INSTICC. SciTePress, 2023, pp. 412–419. ISBN: 978-989-758-647-7. DOI: 10.5220/0011849200003464.

[54] Hendrik Winkelmann and Herbert Kuchen. “Symbolic Execution of NoSQL Applications Using Versioned Schemas”. In: *Proceedings of the 36th Annual ACM Symposium on Applied Computing*. SAC ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 1778–1787. ISBN: 9781450381048. DOI: 10.1145/3412841.3442050. URL: https://doi.org/10.1145/3412841.3442050.

[55] Hendrik Winkelmann, Laura Troost, and Herbert Kuchen. “Constraint-Logic Object-Oriented Programming for Test Case Generation”. In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. SAC ’22. Virtual Event: Association for Computing Machinery, 2022, pp. 1499–1508. ISBN: 9781450387132. DOI: 10.1145/3477314.3507015. URL: https://doi.org/10.1145/3477314.3507015.

[56] Yunhui Zheng et al. “Z3str2: an efficient solver for strings, regular expressions, and length constraints”. In: *Formal Methods in System Design* 50 (2017), pp. 249–288.

Working Papers, ERCIS

Nr. 1 Becker, J.; Backhaus, K.; Grob, H. L.; Hoeren, T.; Klein, S.; Kuchen, H.; Müller-Funk, U.; Thonemann, U. W.; Vossen, G.; European Research Center for Information Systems (ERCIS). Gründungsveranstaltung Münster, 12. Oktober 2004.

Nr. 2 Teubner, A.: The IT21 Checkup for IT Fitness: Experiences and Empirical Evidence from 4 Years of Evaluation Practice. 2005.

Nr. 3 Teubner, A.; Mocker, M.: Strategic Information Planning – Insights from an Action Research Project in the Financial Services Industry. 2005.

Nr. 4 Gottfried Vossen, Stephan Hagemann: From Version 1.0 to Version 2.0: A Brief History Of the Web. 2007.

Nr. 5 Hagemann, S.; Letz, C.; Vossen, G.: Web Service Discovery – Reality Check 2.0. 2007.

Nr. 6 Teubner, A.; Mocker, M.: A Literature Overview on Strategic Information Management. 2007.

Nr. 7 Ciechanowicz, P.; Poldner, M.; Kuchen, H.: The Münster Skeleton Library Muesli – A Comprehensive Overview. 2009.

Nr. 8 Hagemann, S.; Vossen, G.: Web-Wide Application Customization: The Case of Mashups. 2010.

Nr. 9 Majchrzak, T.; Jakubiec, A.; Lablans, M.; Ükert, F.: Evaluating Mobile Ambient Assisted Living Devices and Web 2.0 Technology for a Better Social Integration. 2010.

Nr. 10 Majchrzak, T.; Kuchen, H: Muggl: The Muenster Generator of Glass-box Test Cases. 2011.

Nr. 11 Becker, J.; Beverungen, D.; Delfmann, P.; Räckers, M.: Network e-Volution. 2011.

Nr. 12 Teubner, A.; Pellengahr, A.; Mocker, M.: The IT Strategy Divide: Professional Practice and Academic Debate. 2012.

Nr. 13 Niehaves, B.; Köffer, S.; Ortbach, K.; Katschewitz, S.: Towards an IT consumerization theory: A theory and practice review. 2012

Nr. 14 Stahl, F., Schomm, F., Vossen, G.: Marketplaces for Data: An initial Survey. 2012.

Nr. 15 Becker, J.; Matzner, M. (Eds.).: Promoting Business Process Management Excellence in Russia. 2012.

Nr. 16 Teubner, A.; Pellengahr, A.: State of and Perspectives for IS Strategy Research. 2013.

Nr. 17 Teubner, A.; Klein, S.: The MÃ¼nster Information Management Framework (MIMF). 2014.

Nr. 18 Stahl, F.; Schomm, F.; Vossen, G.: The Data Marketplace Survey Revisited. 2014.

Nr. 19 Dillon, S.; Vossen, G.: SaaS Cloud Computing in Small and Medium Enterprises: A Comparison between Germany and New Zealand. 2015.

Nr. 20 Stahl, F.; Godde, A.; Hagedorn, B.; Köpcke, B.; Rehberger, M.; Vossen, G.: Implementing the WiPo Architecture. 2014.

Nr. 21 Pflanzl, N.; Bergener, K.; Stein, A.; Vossen, G.: Information Systems Freshmen Teaching: Case Experience from Day One (Pre-Version of the publication in the International Journal of Information and Operations Management Education (IJIOME)). 2014.

Nr. 22 Teubner, A.; Diederich, S.: Managerial Challenges in IT Programmes: Evidence from Multiple Case Study Research. 2015.

Nr. 23 Vomfell, L.; Stahl, F.; Schomm, F.; Vossen, G.: A Classification Framework for Data Marketplaces. 2015.

Nr. 24 Stahl, F.; Schomm, F.; Vomfell, L.; Vossen, G.: Marketplaces for Digital Data: Quo Vadis?. 2015.

Nr. 25 Caballero, R.; von Hof, V.; Montenegro, M.; Kuchen, H.: A Program Transformation for Converting Java Assertions into Controlflow Statements. 2016.

Nr. 26 Foegen, K.; von Hof, V.; Kuchen, H.: Attributed Grammars for Detecting Spring Configuration Errors. 2015.

Nr. 27 Lehmann, D.; Fekete, D.; Vossen, G.: Technology Selection for Big Data and Analytical Applications. 2016.

Nr. 28 Trautmann, H.; Vossen, G.; Homann, L.; Carnein, M.; Kraume, K.: Challenges of Data Management and Analytics in Omni-Channel CRM. 2017.

Nr. 29 Rieger, C.: A Data Model Inference Algorithm for Schemaless Process Modeling. 2016.
Nr. 30 Bünder, H: A Model-Driven Approach for Graphical User Interface Modernization Reusing Legacy Services. 2019.
Nr. 31 Stockhinger, J.; Teubner, R.: How Digitalization Drives the IT/IS Strategy Agenda. 2020.
Nr. 32 Dageförde, J. C.; Kuchen, H.: Free Objects in Constraint-logic Object-oriented Programming. 2020.
Nr. 33 Plattfaut, R.; Coners, A.; Becker, J.; Vollenberg, C.; Koch, J.; Godefroid, M.; Halbach-Türscherl, D.: Patient Portals in German Hospitals – Status Quo and Quo Vadis. 2020.
Nr. 34 Teubner, R.; Stockhinger, J.: IT/IS Strategy Research and Digitalization: An Extensive Literature Review. 2020.
Nr. 35 Distel, B.; Engelke, K.; Querfurth, S.: Trusting me, Trusting you – Trusting Technology? A Multidisciplinary Analysis to Uncover the Status Quo of Research on Trust in Technology. 2021.
Nr. 36 Becker, J.; Distel, B.; Grundmann, M.; Hupperich, T.; Kersting, N.; Löschel, A.; Parreira do Amaral, M.; Scholta, H.: Challenges and Potentials of Digitalisation for Small and Mid-sized Towns: Proposition of a Transdisciplinary Research Agenda. 2021.
Nr. 37 Lechtenberg, S.; Hellingrath, B.: Applications of Artificial Intelligence in Supply Chain Management: Identification of main Research Fields and greatest Industry Interests. 2021.
Nr. 38 Schneid, K.; Di Bernardo, S.; Kuchen, H.; Thöne, S.: Data-Flow Analysis of BPMN-Based Process-Driven Applications: Detecting Anomalies across Model and Code. 2021.