Automated Instantiation of Control Flow Tracing Exercises*

Clemens Eisenhofer
TU Wien
Vienna, Austria
clemens.eisenhofer@tuwien.ac.at

Martin Riener
TU Wien
Vienna, Austria
martin@derivation.org

One of the first steps in learning how to program is reading and tracing existing code. In order to avoid the error-prone task of generating variations of a tracing exercise, our tool TATSU generates instances of a given code skeleton automatically. This is achieved by a finite unwinding of the program in the style of bounded model checking and using the SMT solver Z3 to find models for this unwinded program.

1 Introduction

Amongst the aims of introductory programming courses is to teach the understanding of small programs. Lister et al. [12, 8] argue that students are often able to trace a program before they can formulate its purpose. This indicates that tracing exercises help students to reach an operational phase where they can reason about their programs. Lister also proposes multiple choice questions [7] that ask the students to identify the output of a small program correctly. Other exercises require the students to identify which code snippets should fill a hole in a program skeleton such that it follows a given specification or ask whether some proposed modifications of a given program lead to an error.

Even when an introductory programming course does not fully subscribe to this "trace before writing your own code" view, it can complement other teaching methods. The authors are involved in such a course for computer scientists on TU Wien with about 650 students. There, tracing questions are used for practice, but also contribute 20% to the overall grade in form of an online assessment within TUWEL, the local Moodle instance. The larger part of the course is spent in lab sessions explaining self-written programs to the instructor and the group.

Since the high number of students requires an adequate pool of questions we create variations of a single question by instantiating a code schema with different values. Historically this has been done manually which is error prone. As an alternative these values can be generated with an SMT solver using ideas borrowed from bounded model checking. For TATSU we use the Z3 solver [9] as it supports some non-standardised functionalities required by our tool. In the following we report on the ongoing development of TATSU and its use to generate quizzes. These quizzes contain single/multiple choice questions as well as questions asking for the output or return value of a given function. The program generator is open source and can be downloaded from our GitLab instance[1].

Generating programs for such exercises in this context is quite different from general synthesis or verification tasks: the programs are short and beginners are not confronted with the full syntax of the language. We also focus on questions of the kind "How often is this loop executed?" rather than checking for corner cases like integer overflows. TATSU automatises the task by taking the skeleton of a Java

* We would like to thank Stefan Podlipnig for his continuing support and input on the paper. His lead in the introductory programming course for computer science on TU Wien with his ideas for TATSU made this work possible. We would also like to thank Nikolaj Bjørner for his help with integrating Z3 in TATSU.

[1] `https://git.logic.at/ep1-tools/tatsu-generator`

J. Marcos, W. Neuper and P. Quaresma (Eds.): Theorem Proving Components for Educational Software 2021 (ThEdu'21)
EPTCS 354, 2022, pp. 43–58, doi:10.4204/EPTCS.354.4

© C. Eisenhofer, M. Riener

program together with a set of pre- and post-conditions and generating a set of instances that fulfill these conditions.

2 Overview

Control flow tracing exercises consist of small computer-program sources that can be used to test a student's ability to read and understand given code. Two examples for a control flow tracing exercise in the Java programming language can be found in Figure 1.

Order the following three loops by the number of executions. You can use the counters a,b,c to check how often a loop is executed.

```
int a = 0;
for (int i = 0; i < 16; i += 2) {
    a++; // Loop A
}

int b = 0;
for (int i = 0; i < 17; i += 2) {
    b++; // Loop B
}

int c = 0;
for (int i = 0; i < 13; i += 7) {
    c++; // Loop C
}
```

Select one:

- b > a > c
- a > b > c
- c > a > b
- b > c > a
- a > c > b
- c > b > a

(a)

Which of the values for limit, inc, start lead to different behaviour (different values of a,b) in the two loops?

```
int a = 0;
int b = 0;
int limit = /* ? */;
int inc =   /* ? */;
int start = /* ? */;
for (int i = start; i < limit; i += inc) {
    // Loop A
    if (i % 2 == 0)
        a++;
    if (i % 3 == 0)
        a++;
}

for (int i = start; i < limit; i += inc) {
    // LOOP B
    if (i % 2 == 0)
        b++;
    else if (i % 3 == 0)
        b++;
}
```

Select one or more:

- int limit = 10; int inc = 5; int start = 3;
- int limit = 11; int inc = 3; int start = 3;
- int limit = 14; int inc = 3; int start = 3;
- int limit = 17; int inc = 5; int start = 1;
- int limit = 15; int inc = 5; int start = 3;
- int limit = 18; int inc = 2; int start = 3;

(b)

Figure 1: Multiple choice question asking to count loops (a) and to distinguish successive conditionals from an if-then-else (b).

In general, a tracing question requires the student to find the values of a particular set of variables or what output the given program has. Tracing may also be used indirectly: the counters in Figure 1(a) correspond to the number of times a loop is executed, but the student only needs to order the loops by their execution length.

In a similar spirit, the program in Figure 2(b) tests the understanding of successive and nested branching. In cases where the loop variable is divisible by 6 the two loops behave differently. The student receives a list of initialization blocks and has to mark those where the divergence results in a different computation between the programs.

To generate variations of these exercises, we annotate the skeleton of a Java program with constraints on the values of variables and assertions over some values. Often the assertions express a post-condition of the whole program, but they may be inserted in between statements as well. Figure 2 shows the skeletons used to generate the questions above.

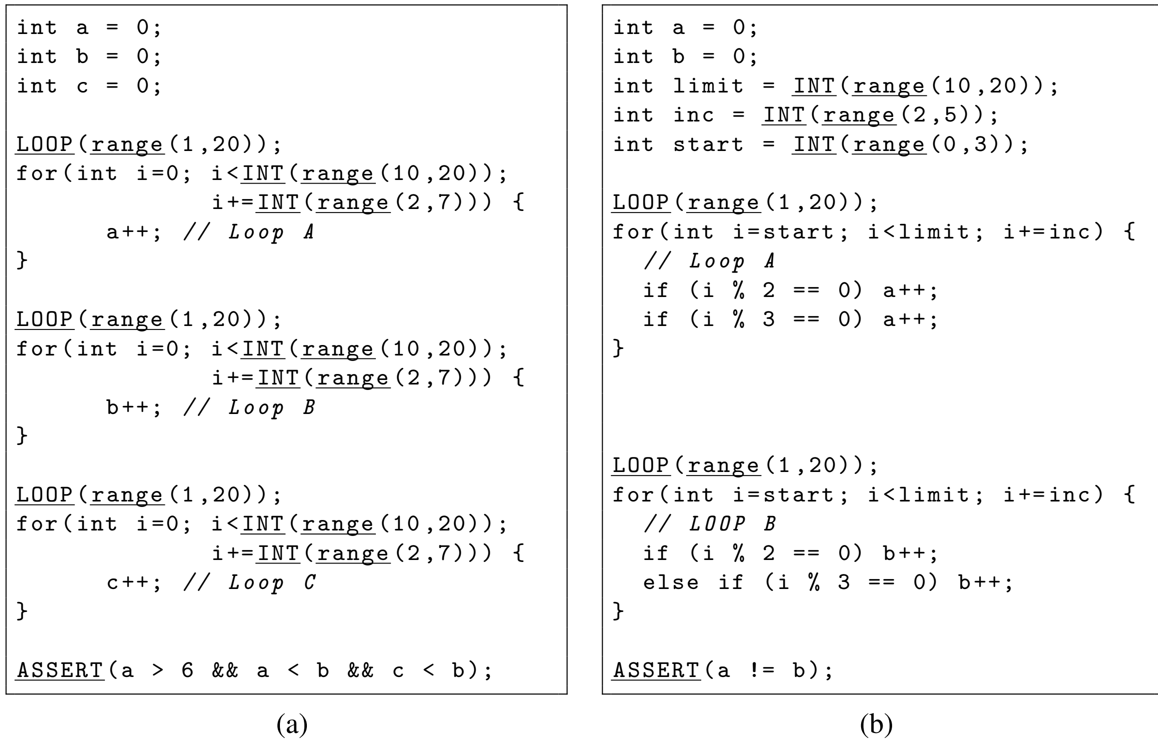


Figure 2: Tracing exercises focusing on higher-level analysis of a program

3 Question Types

Classically, a tracing question has a text field where students enter the computed result. The validation of the answer is problematic though. The input needs to be validated due to ambiguous solutions (e.g. `0x0F` and `15` may be used interchangeably). While this problem might still be avoided by adjusting the exercise, the way e-learning platforms like Moodle validate free-form exercises by comparing the answer to a given list of possible solutions is quite limiting. [2] For example, a task asking to fill in the missing part of a program can not be expressed due to the unbounded number of possible code-snippets that could be entered. In general, questions that have an infinite set of possible answers (e.g. where any string of an appropriate length is a correct solution) lead to similar problems. For this reason, we focus on questions where all solutions can be pre-computed. Often a subset of these solutions is presented in a multiple-choice form but under these limitations, free-form answers are also possible.

Another issue is that code can be easily copy pasted into an IDE and run instead of manually computing the result. Turning the code fragment into an image makes this approach harder but not impossible: a student might use OCR software to recover the text or they might simply type it into an editor.

Alternatively, we could adapt the question by asking the student to analyze multiple code fragments. Instead of computing the result they select those fragments that exhibit a particular behaviour. A possible question could propose multiple implementations for binary search and ask the students to select the

[2] This is mostly a security requirement of the platform because general input validation would need the execution of arbitrary programs. The Moodle plugin Coderunner (`https://moodle.org/plugins/qtype_coderunner` implements a secure environment for such tasks but it is not part of the default distribution. Often the installation of such a plugin is out of the control of the teaching such that we can not expect it to be present.

Given	Possible Question
A complete code fragment	"What is the result/output of the given code?"
	"Does the result/output change if we replace line ... by ... ?"
A set of complete code fragments	"Which of the given code fragments produces the result/output ... ?"
	"Which of the given code fragments produce the same (concrete) results/outputs ... ?"
A code fragment containing a hole (`??`)	"Which value must be inserted instead of `??` such that the loop iterates ... times/the result is ... ?"

Figure 3: Possible questions

correct ones.

Since correctness proofs are undecidable in general, a BMC based approach is likely to fail for such questions. But the example in Figure 2(b) presents a compromise: we leave holes for literals marked by `??` in the program and ask for values that lead to different behaviour of the two loops. The answer candidates are generated in two runs, the first creates the correct solutions and the second the wrong solutions.

The table in figure 3 gives an overview of the question types we consider.

4 Specifying a Program Skeleton

The exercises are intended for beginners knowing only a restricted subset of the Java programming language. Therefore we restrict ourselves to what Lister calls a "Pascal-like" language with conditionals (if-else, ternary operator), loops (while, do-while, for), restricted jump statements (return, break, continue), and function calls (including recursion) as control flow operators. Data types are usually booleans, integers and arrays of integers. Further types the tool can deal with are strings, one dimensional string arrays and two dimensional integer arrays. However, reasoning about these kind of datatypes is often hard for solvers and the computation time consequently quite high for larger code-snippets.

A program skeleton is a syntactically correct Java program with additional functions and annotations with special meaning. To differentiate the two we underline the additional functions in our examples. Processing the template in TATSU results in one or multiple instances that fulfill the expressed constraints. An instance is a plain Java program without any annotations left.

The most important functions are the placeholders that represent concrete constant values in the generated instances. The placeholder itself represents the type (`INT`, `BOOLEAN`, `INTARRAY`, etc.) and describes the possible values via an argument that contains a constraint. The function `list` selects one of its arguments as possible value which is particularly suited for non-contiguous intervals. The function `range` stands for an integer value in the range of its arguments. For example, the loop bounds and increments in Figure 2(a) are specified as integers in the range of $[10;20]$ and $[2;7]$ respectively. In Figure 4 the placeholder `INTARRAY` represents an `int[]` of size 12 containing values in the range $[-25;25]$.

Furthermore, we can also restrict the values for the placeholders indirectly by adding assertions with `ASSERT`. In Figure 4 we have two such assertions: the one in `mystery` guarantees that the difference of maximum and the minimum in each recursive step is at least 10. The assertion in `start` ensures that the

```
@MAIN
static int[] start() {
  int[] input = INTARRAY(list(12), range(-25, 25));
  ASSERT(__distinct(input, 12));
  return mystery(input, 0, input.length - 1);
}

@REC(5)
static int[] mystery(int[] data, int from, int to) {
  if (from == to) return new int[] { data[from], data[from] };

  int[] left = mystery(data, from, (from + to) / 2);
  int[] right = mystery(data, (from + to) / 2 + 1, to);

  int r[] = new int[2];
  r[0] = left[0] < right[0] ? left[0] : right[0];
  r[1] = left[1] < right[1] ? right[1] : left[1];
  ASSERT(Math.abs(r[0] - r[1]) > 10);
  return r;
}
```

Figure 4: A recursion that computes minimum and maximum of a given input.

generated array does not contain duplicate elements by using the helper function `__distinct`.

Other important constructs are the `LOOP` and `@REC` annotations which limit the unwinding of loops and recursions. This is motivated by our goal of translating into a decidable (quantifier-free) logic theory suited to bounded model checking[3]. The `LOOP` annotation describes the number of iterations a loop may be repeated. This is often a range with lower bound zero and a reasonable upper bound. However, to guarantee a minimum complexity for a student to trace, a different lower bound on the interval or even a fixed list of possible repetitions may be specified. For example, the loops in Figure 2 need to iterate at least once and up to twenty times.

The `@REC` annotation describes the maximum amount of recursive calls of a function for the current trace. In contrast to `LOOP` the `@REC` annotation is optional and defaults to non-recursive calls only. For example, the maximal recursion depth of `mystery` in Figure 4 is set to five.

These limits can be thought of as constraints on the desired models. Similar to overly strict assertions, TATSU will simply report that it could not find a model in these cases.

Assertions that can not be represented by a single statement/assertions can be wrapped into an `ASSERTBLOCK`. This statement indicates that the following block should be considered as a "large assertion" and not as code that will be handed out to the students.

The skeleton contains two different kinds of code: code that is present in the generated snippet and code that is only considered during generation. An example can be seen in Figure 5. The `ASSERTBLOCK` there constraints the array `arr` such that all values but one are odd. This block is not suitable to be used as hole in a question but it improves the examples generated.

In a limited number of cases we can also use invariant-assertions to generate instances that can take a large number of loop iterations into account. Consider the skeleton shown in Figure 6. Again, the actual value for `INT(range(1, 100))` is replaced by `??` and the student is asked to find a value such that the

[3] The generated problem might still be undecidable due to the assertions. Examples for this are constraints on strings or when explicitly using quantifiers but this is under control of the user.

```
int[] arr = INTARRAY(list(5), range(1, 100));
int idx = INT(range(0, 4));
ASSERTBLOCK();
{
  ASSERT((arr[idx] % 2) == 0);
  ASSERT(__distinct(arr, 5));
  LOOP(list(5));
  for (int i = 0; i < arr.length; i++) {
    ASSERT(__impl(i != idx, (arr[i] % 2) == 1));
  }
}
arr[idx] /= 2;
arr[idx] *= 2;
```

```
int[] arr = new int[] { 23, 8, 43, 67, 59 };
int idx = 1;
arr[idx] /= 2;
arr[idx] *= 2;
```

Figure 5: Large assertions

last element in the array has the value given in `res`. As the number of iterations is large students cannot simply trace the flow directly. They have to find out that in the ith iteration of the loop the value in the current array element is $inc \cdot i$ and that after `arr.length` iterations the value at the last array position is $inc \cdot (arr.length - 1)$. Internally, the invariant replaces the loop unwinding but adds the assertion that the invariant follows from the loop body. This approach is helpful in cases with very high loop counts that create enormous translations which would not terminate in an acceptable amount of time. In general the invariant may be an under-approximation of the program state after the loop. To avoid spurious models the user has to be very careful to specify a sufficiently strong invariant. In this case the safety net of running the generated program in Java described at the end of Section 5 becomes obligatory.

5 Implementation

TATSU shares some ideas with automated unit test generators like KLEE [5] or Java PathFinder [10, 13]. There, we want to find inputs for a given program that should trigger as many different program behaviours as possible whereas we create multiple programs that should behave quite similar. The strongest influence are programs like CBMC [1, 6] where bounded model checking is used to find bugs in programs. Where CBMC usually checks all possible program traces, we are only interested in generating some witness traces to fill the holes in the program skeleton. Nevertheless, the approach is similar in the sense that it finitely unwinds the program into a variant of a single static assignment form (SSA form) and uses a solver to find a model of the logical representation. The model values are used to fill the placeholders in the program skeleton. Finally, we run the instantiated skeleton to check if none of the assertions is violated in an actual Java environment.

Our encoding of Java structures into first-order terms is based on a simplified Java specification. A value of boolean type maps to a propositional variable, integer types map to bitvectors (Figure 7). Java strings values map to SMTLib strings. This, on the one hand, gives us the advantage of arguing about strings of an arbitrary size (as this requires no loop unwinding) but, on the other hand, might increase

```
int[] arr = new int[INT(range(6000, 10000))];
int inc = INT(range(1, 100));
int i = 1;
INVARIANT(i >= 1 && i <= arr.length && arr[i - 1] == (i - 1) * inc);
while (i < arr.length) {
  arr[i] = arr[i - 1] + inc;
  i++;
}
System.out.print(arr[arr.length - 1] == INT(range(60000, 100000)));
ASSERT(__out.equals("true"));
```

```
int[] arr = new int[9546];
int inc = 8;
int i = 1;
while (i < arr.length) {
  arr[i] = arr[i - 1] + inc;
  i++;
}

System.out.print(arr[arr.length - 1] == 76360);
```

Figure 6: Invariant assertions

complexity quite a lot.

Java type	boolean	byte	short	int	long	char	String
Placeholder	BOOLEAN			INT		CHAR	STRING
SMTLib	Bool	BitVec 8	BitVec 16	BitVec 32	BitVec 64	Unicode[4]	String

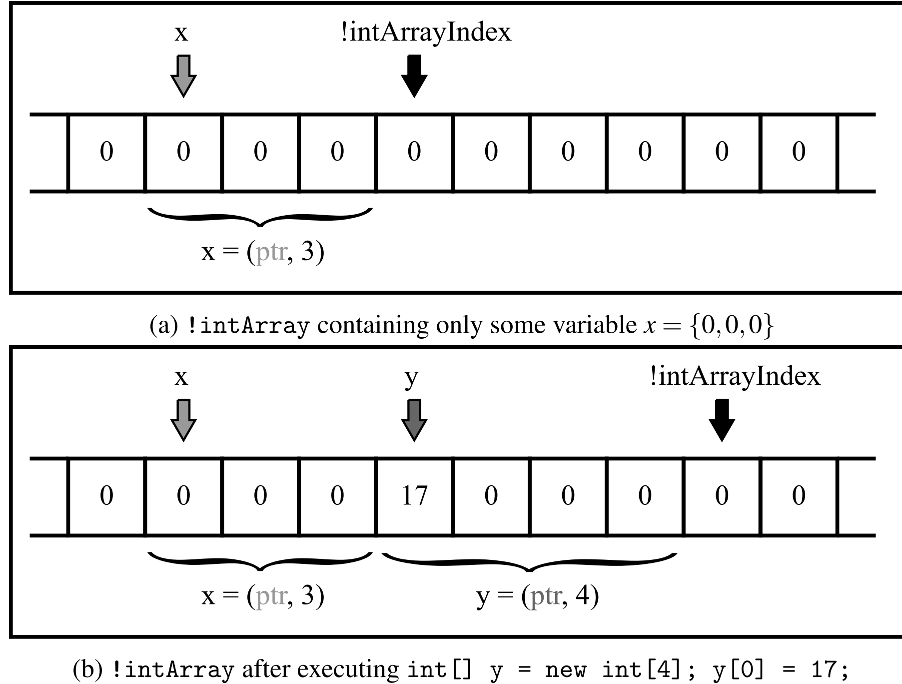
Figure 7: Mapping of Java types to SMTLib types

Arrays are the only supported objects where we need differentiate between object and reference equality, as strings are readonly. Arrays are modeled as subsets of the heap that is a single SMTLib array. A concrete Java array is represented as a pair of the index of the position in the heap and the arrays length. These tuples can be determined during unwinding since Java arrays neither overlap nor are they resizable. The heap is modeled explicitly as we have to allow two array variables to refer to the same memory address and writing to one of them has to change the value of the other one as well. From a logical perspective: For the SMT solver two arrays are considered equal iff their content is equal but in Java two arrays are equal iff the memory address they are referencing is equal. Therefore, for every array type (`int[]`, `int[][]`, `String[]`) there exist global variables (in SSA form) representing the values on the heap for the corresponding type. For example, for `int[]` there exists a constant called `!intArray` that contains all the `int[]` values in the system as a subset. Furthermore, there exists another constant `!intArrayIndex` that marks the index in the global int array where arrays generated by `new int[y]` should be located. i.e., it represents the address of the generated array. This value will be incremented after the allocation accordingly to point to a free position again. Monotonically incrementing this index is a simple strategy that allows us to simulate allocating memory on the heap, however, forces us keeping track of variables that might have been already freed by Java. A concrete example how arrays are

[4] Z3 specific

allocated is shown in Figure 8.

In case of multidimensional (“jagged”) arrays like `int[][]` we can make use of multiple global arrays. One refers to indices in the global `int[]` array and one contains the lengths of the corresponding arrays. The mapping of the 3 implemented array types is presented in Figure 9.



(a) `!intArray` containing only some variable $x = \{0,0,0\}$

(b) `!intArray` after executing `int[] y = new int[4]; y[0] = 17;`

Figure 8: Allocating memory

Java type	int[ ]	int[ ][ ]	String[ ]
Placeholder	INTARRAY	INT2DARRAY	STRINGARRAY
SMTLib	(BitVec 32, BitVec 32)		
	Array BitVec 32	3 · (Array BitVec 32)	Array String

Figure 9: Mapping of Java array-types to SMTLib types

As in SMTLIB we use multi-sorted first-order logic with equality and if-then-else conditionals [3]. The theories used are bitvectors (BV), extensional arrays (AX), strings (S), datatypes (DT) [2, 3]. The standard translation is quantifier-free but quantifiers can be added explicitly by the user if required (e.g., as part of invariants etc.). We also rely on custom extensions only present in Z3 (conversion between strings and bitvectors, constant arrays, etc.). A more formal definition of this section can be found in Appendix A.

Our tool converts the program’s structure into SSA form. In order to do so, every possible change of the value of a variable requires us to introduce a new SMT constant. In order to do so, Java variables `x` are represented by a sequence of constants `x@[version]@[writecount]` where `[writecount]` is a number starting with 0 that will be incremented every time there is an assignment to the variable `x`. As multiple different variables with the same name `x` might occur during translating the code (due to Java scoping, loop unwinding, function inlining, ...) the number `[version]` represents how many different

variables with the same name have been processed before. Most SMT constants used internally include some characters like “@” or “!” to avoid collisions with user-defined variables. However, in some cases it is not possible as the internal variables needs to be reflected in the Java code. An example for such a name is the pseudo-variable `__out` representing the content printed to console so far. It can be used in assertions to constraint the output and therefore needs a Java-compatible name. The output command `System.out.print(s);` is then simply modeled by `__out += s;`.

Among the control constructs, the `if-then-else` conditional translates to two logical implications expression whereas `(do-)while`- and `for`-loops are removed during unwinding. The `break`, `continue`, and `return` statements are also handled directly during the SSA form generation by a program transformation.

In order to introduce only as few additional SSA-constants as possible we slightly modify the SSA approach. The idea of the SSA-form is that every variable assignment introduces a new constant [11]. In case of branching (due to a simple if-then-else or introduced by loop unwinding) the ordinary approach would use disjoint constants in both branches. However, the approach used here makes only sure that the solver does not have to assign different values to the same constant. Some SMT constant `x` might, therefore, occur in both the then- and the else-branch. Figure 10 shows both approaches.

Java code	Ordinary SSA	Modified SSA

int x = init;
if (cond) {
x += v1;
x −= v2;
} **else** {
x ∗= v3;
}
int y = x + 1;
	x@0 = init;
	x@1 = x@0 + v1;
	x@2 = x@1 − v2;
	x@3 = x@0 ∗ v3;
	x@4 = cond ? x@2 : x@3;
	y@0 = x@4 + 1;
		x@0 = init;
		if (cond) {
		x@1 = x@0 + v1;
		x@2 = x@1 − v2;
		} **else** {
		x@1 = x@0 ∗ v3;
		x@2 = x@1;
		}
		y@0 = x@2 + 1;

	(and (= x@0 init) (= x@1 (+ x@0 v1)) (= x@2 (− x@1 v2)) (= x@3 (∗ x@0 v3)) (= x@4 (ite cond x@2 x@3)) (= y@0 (+ x@4 1)))	(and (= x@0 init) (ite cond (and (= x@1 (+ x@0 v1)) (= x@2 (− x@1 v2))) (and (= x@1 (∗ x@0 v3)) (= x@2 x@1))) (= y@0 (+ x@2 1)))

Figure 10: SSA translation

As we can see, our representation is slightly larger in general than the ordinary one (w.r.t. SMTLib

code size). However, it has two advantages over its alternative. Firstly, it introduces less constants (especially if both branches contain multiple write accesses to the same variable). This allows us to directly read off the concrete data-flow from the generated model which is particular interesting in potential post-processing steps. Secondly, we preserve the original structure of the Java code which can be useful in case the generated problem encoding has to be modified afterwards.

Post-processing, maintainability, and readability play an important role in the underlying encoding for us. Transforming and simplifying the generated code appears to be a reasonable idea. Our code optimizations lead from simple arithmetic pre-computations (for example, replacing $1+2$ by 3) to keep track of possible values of variables via an interval abstraction. Doing these simplifications can be used to eliminate unreachable branches or to detect better loop-unwinding bounds. An example for the latter can be found in Figure 11 where the loop limit is signifcantly larger than required: The variable `count` might be initialized by some value in the range $[0,10]$ and `i` has some value in the range $[3,5]$. Iterating one time will result in `count` being contained in $[1,11]$ and `i` between $[5,7]$. We can update the intervals until the loop condition is false or the regular upper limit is reached. In our example, this happens after the seventh iteration where the interval $[8,18]$ for `count` and the interval $[19,21]$ for `i` do not intersect anymore. From this point on, unwinding the loop further does not make sense because at least at this point the loop condition is invalidated.

```
int i = INT(range(3, 5));
int count = INT(range(0, 10));
LOOP(range(0, 100000));
while (i < count) {
   count++;
   i += 2;
}
```

Figure 11: Detection of loop bounds by unwinding.

In ordinary software there are mostly no (reasonable low) loop iteration bounds. However, when considering small exercises for students with bounded constants we can often completely unwind loops. None of these simplifications can be understood as an assistance for the solver. Most solvers have very sophisticated preprocessors and would probably do much more powerful optimizations on their own. We ran all our Unit-Tests with and without simplification. There was at most only a very slight performance improvement when optimization was turned on. But this was not the point of doing these kind of simplifications. The main reason for applying them is to generate a short, succinct, and as readable as possible representation of the input program. Apart from that, is mostly easier and more convenient for persons writing program skeletons to let the program detect the loop bounds itself instead of estimating them by their own unless they explicitly want to control the number of iterations. The creators of the program skeletons should only have to get in touch with the underlying technology/logic to the extent that is absolutely necessary. The tool should not require its users to understand what is happening internally.

We then use the SMT solver Z3 to find models of the generated formula. Each model value is then translated back to a Java expressions. The conversion of booleans, characters, integer, and string types back to Java is (almost) direct whereas arrays are constructed as sub-arrays of the heap. These expressions are inserted for the placeholders of the skeleton to create the Java instance. Multiple instances can be generated by excluding the placeholder values found so far by adding appropriate blocking clauses.

As a safety net we double check the programs produced by adding the skeleton's constraints as Java

assertions and running the generated instance. This provides a second source for both the output and return values of the computation. Doing this check also gives us a method to guarantee the correctness of our found instance and to detect bugs in the generator may it be due to translation mistakes or a different understanding of the Java specification. (The behaviour of the Java standard library may change between versions[5].)

6 Outlook and Limits of the Method

We are currently experimenting with further optimizations that can reduce generation time. In particular code doing manipulations on (nearly) all elements of an array (e.g., sorting) is very hard to deal with w.r.t. performance. We consider floating-point types out of reach both due to the difficulty of their logical encoding [4] and the limited contribution to understanding precise float computation by tracing. Furthermore, the generator currently misses object oriented aspects like objects and class variables.

A theoretical problem that might occur is that we cannot guarantee that all possible models have the same probability of getting chosen. We do not really have much control over what model is found by the solver and it might happen that the solver yields very similar models every time. For example, the expression `INT(range(0, 1000))` might become 0 in the first model, then 1, then 2, and so on. In practice, we have not encountered such strong dependencies between the generated models. However, this depends on the internals of the used solver.

References

[1] Alessandro Armando, Jacopo Mantovani & Lorenzo Platania (2009): *Bounded model checking of software using SMT solvers instead of SAT solvers*. *Int. J. Softw. Tools Technol. Transf.* 11(1), pp. 69–83, doi:10.1007/s10009-008-0091-0.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli: *SMTLib 2.6 Theory Definitions*. Available at `https://smtlib.cs.uiowa.edu/theories.shtml`.

[3] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2021): *The SMT-LIB Standard: Version 2.6*. Technical Report, Department of Computer Science, The University of Iowa. Available at `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2021-05-12.pdf`.

[4] Martin Brain, Cesare Tinelli, Philipp Rümmer & Thomas Wahl (2015): *An Automatable Formal Semantics for IEEE-754 Floating-Point Arithmetic*. In: *22nd IEEE Symposium on Computer Arithmetic*, IEEE, pp. 160–167, doi:10.1109/ARITH.2015.26.

[5] Cristian Cadar, Daniel Dunbar & Dawson R. Engler (2008): *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs*. In Richard Draves & Robbert van Renesse, editors: *8th USENIX Symposium on Operating Systems Design and Implementation*, USENIX Association, pp. 209–224. Available at `http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf`.

[6] Edmund M. Clarke, Daniel Kroening & Flavio Lerda (2004): *A Tool for Checking ANSI-C Programs*. In Kurt Jensen & Andreas Podelski, editors: *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, *Lecture Notes in Computer Science* 2988, Springer, pp. 168–176, doi:10.1007/978-3-540-24730-2_15.

[5] For example the expression `"" == "abc".substring(0,0)` evaluates to false in OpenJDK prior to version 15, but it evaluates to true in current versions. This is due to an optimization of the substring function made in the bug report JDK-8240094 (`https://bugs.java.com/bugdatabase/view_bug.do?bug_id=JDK-8240094`).

[7] Raymond Lister (2001): *Objectives and objective assessment in CS1*. In Henry MacKay Walker, Renée A. McCauley, Judith L. Gersting & Ingrid Russell, editors: *Proceedings of the 32rd SIGCSE Technical Symposium on Computer Science Education*, ACM, pp. 292–296, doi:10.1145/364447.364605.

[8] Raymond Lister (2011): *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. In: *Proceedings of the Thirteenth Australasian Computing Education Conference - Volume 114*, ACE '11, Australian Computer Society, Inc., AUS, pp. 9–18.

[9] Leonardo de Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, Berlin, Heidelberg, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[10] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person & Mark Pape (2008): *Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing Nasa Software*. ISSTA'08, Association for Computing Machinery, New York, NY, USA, pp. 15–26, doi:10.1145/1390630.1390635.

[11] Barry K. Rosen, Mark N. Wegman & F. Kenneth Zadeck (1988): *Global Value Numbers and Redundant Computations*. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, Association for Computing Machinery, New York, NY, USA, pp. 12–27, doi:10.1145/73560.73562.

[12] Donna Teague & Raymond Lister (2014): *Blinded by their Plight: Tracing and the Preoperational Programmer*. In Benedict du Boulay & Judith Good, editors: *Proceedings of the 25th Annual Workshop of the Psychology of Programming Interest Group*, Psychology of Programming Interest Group, p. 8. Available at `http://ppig.org/library/paper/blinded-their-plight-tracing-and-preoperational-programmer`.

[13] Willem Visser, Corina S. Pasareanu & Sarfraz Khurshid (2004): *Test Input Generation with Java PathFinder*. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, Association for Computing Machinery, New York, NY, USA, pp. 97–107, doi:10.1145/1007512.1007526.

A Java Translation to FOL

In the following we will describe a formal translation of the input program into a formula. The actual implementation (SSA, constant naming, etc.) of the translation is outlined in the main part of this paper. We restrict the formalisation to a smaller subset of the supported Java fragment due to space limitations. A translation of Java variables to a logical constants needs to take two issues into account: assignments require the introduction of a fresh successor and the same name might refer to a different memory location in different scopes. A context collects the information necessary to perform this mapping.

Definition 1. A *context* is a tuple $\langle N, V, E \rangle$ where N is a set of logical constants, V is a sequence of mappings from Java variables to logical constants, E is a set of logical formulas.

The set N tracks the logical constants introduced. A fresh constant is then just one that has not been introduced yet. The sequence of mappings $V = [v_1, \ldots, v_n]$ represents the nesting of scopes where the innermost nesting has the highest index. A lookup successively widens the scope until it finds a mapping for the variable in question. A variable declaration updates the innermost context. Entering a new scope adds a fresh mapping to the sequence whereas leaving the scope removes the the last mapping from V. The set E will track some side conditions such the handling of division by zero during the translation of expressions. We will also use the notion $\mathscr{D}(f)$ for the domain of the function f here.

This is formally expressed as follows:

Definition 2. Let $C = \langle N, V, E, P \rangle$ be a context with $V = [v_1, \ldots v_n]$. Then the function $lookup(C, jv)$ maps a a context and a Java variable to a logical constant and is recursively defined as:

$$lookup(C, jv) = \begin{cases} v_n(jv) & \text{if } jv \in \mathscr{D}(v_n) \\ lookup(\langle N, [v_1, \ldots, v_{n-1}], E, P \rangle, jv) & \text{if } jv \notin \mathscr{D}(v_n) \text{ and } n \geq 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Furthermore, let $x \notin N$ be a logical constant, $N' = N \cup \{x\}$, $v'_n(j) = \begin{cases} x & \text{if } x = jv \\ v_n(j) & \text{otherwise} \end{cases}$ be a variable mapping extended by a Java variable jv, and let v_{n+1} be a function with an empty domain.

Then we define the following functions:

$update(C, jv)$	$= \langle N', [v_1, \ldots, v'_n], E, P \rangle$	maps a context and a Java variable to a new context
$enter(C)$	$= \langle N, [v_1, \ldots, v_n, v_{n+1}], E, P \rangle$	maps a context to a new context
$leave(C)$	$= \langle N, [v_1, \ldots, v_{n-1}], E, P \rangle$	maps a context to a new context
$addconstr(C, F,)$	$= \langle N, V, E \cup F, P \rangle$	maps a context and a formula to a new context

We also extend $update(C, JVs)$ to sets of Java variables JV by successively applying $update(C, jv)$ for each $jv \in JVs$. All three functions are undefined for an empty sequence V.

During the unwinding of a Java program we never rely on one of the functions being undefined. This is due to the following invariants of Java programs:

- Program execution always happens within a scope
- Every opened scope is closed again
- Variables have to be assigned before use

The following translation makes some further assumptions:

- `for` and `do-while` loops have been rewritten to `while` loops

- Function bodies have been rewritten such that `return` only occurs as the last statement
- Loop bodies have been rewritten such that `continue` and `break` do not occur anywhere
- The variables `__out` and `__ret` are never declared
- Recursion, strings and arrays have been left out for simplicity
- There are no uninitialised variables

We can now define the translation (including the unwinding of loops) of expressions (T_E) and of statements (T_S). Both functions return a tuple with a context and a logical expression. In the case of T_S the expressions are always of type *Bool* such that they can be used in logical formulas.

Definition 3. We define the functions $T_E(C, expr)$ and $T_E(C, stmt)$ by mutual recursion based on the structure of *expr* and *stmt*.
For T_E We distinguish on the shape of *expr*:

- *expr* is a literal of types *char*, *byte*, *short*, *int*, *long*: $T_E(C, expr) = \langle C, num\rangle$ where *num* is the numeral corresponding to expr. The type of *num* is determined by the table in Figure 7.
- *expr* is Java variable *jv*: $T_E(C, expr) = \langle C, lookup(C, jv)\rangle$
- *expr* is a TATSU placeholder $INT(range(lower, upper))$: let
  $C = \langle N, V, E\rangle$
  $c \notin N$ is a fresh logical constant. The type of c is determined by the placeholder (INT, etc.) and the table in Figure 7.
  l is the numeral corresponding to *lower*, u is the numeral corresponding to *upper*
  $C' = \langle N \cup \{c\}, V, E \cup \{l \leq c, c \leq u\}\rangle$
  then $T_E(C, expr) = \langle C', c\rangle$.
- *expr* is one of the arithmetic expressions $s \circ t$ with $\circ \in \{+, -, *, /, <, >, <=, >=, ==, !=\}$: let
  $$\langle C_1, ls\rangle \ = \ T_E(C, s),$$
  $$\langle C_2, lt\rangle \ = \ T_E(C_1, t),$$
  $$F = \begin{cases} lt \not\approx 0 & \text{if } \circ = / \\ \top & \text{if } \circ \in \{+, -, *, <, >, <=, >=, ==, !=\} \end{cases}$$
  then $T_E(C, expr) = \langle addconstr(C_2, \{F\}), ls \oplus lt\rangle$ where $\oplus$ is the logical arithmetic operator corresponding to $\circ$. Note that the comparison operators are expressions of type *Bool* i.e. they are formulas that can be used as constraints.
- *expr* is a pre-increment operator ++a / --a: let
  $C_1 = update(C, a)$
  $lookup(C_1, a) \approx lookup(C, a) \circ 1$ with $\circ \in \{+, -\}$
  then $T_E(C, expr) = T_E(addconstr(C_1, E), a)$.
- *expr* is a post-increment operator a++ / a--:
  $\langle C_1, t\rangle = T_E(C, a)$
  $C_2 = update(C_1, a)$
  $E = lookup(C_2, a) \approx lookup(C_1, a) \circ 1$ with $\circ \in \{+, -\}$
  then $T_E(C, expr) = \langle addconstr(C_2, E), t\rangle$.

- *expr* is a user defined function call $f(arg_1,\dots,arg_n)$ with body B: let

  $\langle C_1,t_1\rangle \;=\; T_E(C,arg_1),$
  $\langle C_2,t_2\rangle \;=\; T_E(C_1,arg_2),$
  $\dots$
  $\langle C_n,t_n\rangle = T_E(C_{n-1},arg_n),$
  $C' = update(update(\dots update(enter(C),arg_1),\dots arg_n),__ret)$
  $\langle C'',F\rangle = T_S(C',B)$

  then $T_E(C,expr) = \langle leave(addconstr(C'',\{F\},)),lookup(C',__ret)\rangle$.

For T_S we distinguish on the shape of *stmt*:

- *stmt* is the empty statement: then $T_S(C,stmt) = \langle C,\top\rangle$
- *stmt* is a TATSU assertion `ASSERT`$(cond)$: then $T_S(C,stmt) = T_E(C,cond)$
- *stmt* is a block $\{stmt\}$: let

  $\langle C',F\rangle = T_S(enter(C),stmt)$

  then $T_S(C,stmt) = \langle leave(C'),F\rangle$
- *stmt* is a return statement `return` *expr*: let

  $\langle C_1,t\rangle = T_E(C,expr)$
  $C_2 = update(C_1,__ret)$

  then $T_S(C,stmt) = addconstr(C_1,lookup(C_2,__ret) \approx t)$
- *stmt* is a sequence of statements $stmt_1;stmt_2$: let

  $\langle C_1,F\rangle = T_S(C,stmt_1)$
  $\langle C_2,G\rangle = T_S(C_1,stmt_2)$

  then $T_S(C,stmt) = \langle C_2,F \wedge G\rangle$
- *stmt* is an assignment $jv = expr$: let

  $\langle C_1,t\rangle = T_E(C,expr)$
  $C_2 = update(C_1,jv)$

  then $T_S(C,stmt) = \langle C_2,lookup(C_2,jv) \approx t\rangle$
- *stmt* is a variable declaration $type\;\; jv = expr$ with $type \in \{char,byte,short,int,long\}$: let

  $\langle C_1,t\rangle = T_E(C,expr)$
  $C_2 = update(C_1,jv)$

  then $T_S(C,stmt) = \langle C_2,lookup(C_2,jv) \approx t\rangle$.
  Note that the only difference to an assignment is that $v_n(jv)$ during $lookup(C,jv)$ would be undefined here.
- *stmt* is a conditional `if` $(cond)\,stmt_1$ `else` $stmt_2$: let

  $\langle C_0,F\rangle = T_S(C,cond)$
  $\langle C_1,G_1\rangle = T_S(C_0,stmt_1)$ with $C_1 = \langle N_1,V_1,E_1\rangle$
  $\langle C_2,G_2\rangle = T_S(C_0,stmt_2)$ with $C_2 = \langle N_2,V_2,E_2\rangle$
  $C_3 = \langle N_1 \cup N_2,V,\{F \Rightarrow E | E \in E_1\} \cup \{\neg F \Rightarrow E | E \in E_2\})\rangle$
  $Changed = \{jv | lookup(C,jv) \neq lookup(C_1,jv) \vee lookup(C,jv) \neq lookup(C_2,jv)\}$

$C_4 = update(C_3, Changed)$

$H_1 = \bigwedge_{jv \in Changed} (F \Rightarrow lookup(C_4, jv) \approx lookup(C_1, jv))$

$H_2 = \bigwedge_{jv \in Changed} (\neg F \Rightarrow lookup(C_4, jv) \approx lookup(C_2, jv))$

then $T_S(C, stmt) = \langle C_4, H_1 \wedge H_2 \rangle$

- *stmt* is a conditional $\texttt{if}\ (cond)\, stmt_1$: then $T_S(C, stmt) = T_S(C, \texttt{if}\ (cond)\, stmt_1\, \texttt{else}\, \{\,\})$
- *stmt* is a loop $\texttt{LOOP}(range(lower, upper)); \texttt{while}\ (cond)\, stmt'$: we distinguish on the values of *lower* and *upper*:
  - $upper = 0$: then $T_S(C, stmt) = T_S(C, \texttt{ASSERT}(!cond))$
  - $0 < lower < upper$: let
    $stmt'' = \texttt{ASSERT}(cond); \texttt{LOOP}(range(lower - 1, upper - 1)); \texttt{while}\ (cond)\, stmt'$
    then $T_S(C, stmt) = T_S(C, stmt'')$
  - $0 = lower < upper$: let
    $stmt'' = \texttt{if}\ (cond)\, \{\texttt{LOOP}(range(0, upper - 1)); \texttt{while}\ (cond)\, stmt'\}$
    then $T_S(C, stmt) = T_S(C, stmt'')$
- *stmt* is a print statement `System.out.print(expr)`:let
  - $\langle C', t \rangle = T_E(update(C, \texttt{__out}), expr)$
  - $C' = \langle N, V, E \rangle$

  then $T_S(C, stmt) = \langle N, V, E \cup \{lookup(C', \texttt{__out}) \approx str.{+}{+}(lookup(C, \texttt{__out}), t)\} \rangle$

Given an initial context $C = \langle \{out\}, [\texttt{__out} \mapsto out], \{out \approx ""\} \rangle$ we can translate the program by evaluating $\langle C', F \rangle = T_S(C, \texttt{main(new String[]\{\})})$ with $C' = \langle N, V, E \rangle$.
The specification is then $F \wedge E$.