Tamarin: Concolic Disequivalence for MIPS

Abel Nieto

University of Waterloo
anietoro@uwaterloo.ca

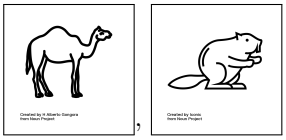
Abstract. TODO

1 Introduction

We are staring at two opaque black boxes laying at our feet. Each box has a narrow slot through which we can place items in the box, but we cannot quite see what is inside. They look approximately like this:

We know each box contains an animal, but we do not know which specific animal is in each one. We would like to find out if both boxes contain the same species of animal. Our solution is simple: we take two carrots, and drop one in each box through the slots.

After a while, a chewing sound emerges from the boxes. We peer into them and, indeed, it looks like the carrots were successfully eaten. Triumphantly, we declare that the boxes contain the same species of animal. The truth is altogether different:



The boxes are assembly programs. The animals are the functions those programs compute. The carrot is unit testing. The task was to determine whether the programs were equivalent. And we failed at it. In this paper, we show a technique that is better than the carrot.

Program equivalence. The program is the specification. The complications of assembly language.

2 Program Equivalence for MIPS

Let us set up the problem a bit more formally. Consider the set P of MIPS-assembly programs that satisfy two restrictions: they take as inputs only the values of registers \$1 and \$2, and when they stop executing we define their

output to be (exclusively) the value of $3. Other side effects, such as printing values to the screen, or system calls, are disallowed.

We can now define a relation $\mathsf{equiv} \subseteq P \times P$ (and its complement, $\not{\mathsf{equiv}}$) of equivalent programs. Given $P_1, P_2 \in P$, we say that P_1 equiv P_2 (read "P_1 is equivalent to P_2") if, for all inputs $1 and $2, one of the following holds:

- Both P_1 and P_2 fail during execution (for example, due to a divide-by-zero error).
- P_1 and P_2 stop with the same output in $3.

For example, the two programs in Figure 1 are equivalent.

```
# P_1
add $3, $1, $2
```

```
# P_2
add $4, $1, $1
lis $5
42
sw $4, 0, $5
add $3, $1, $2
```

Fig. 1. P_1 equiv P_2

Notice that P_1 equiv P_2 even though P_2 modifies the contents of the memory and an additional register ($4), because:

- Both P_1 and P_2 terminate without errors.
- The value of $3 will be the same when they do so.

Unfortunately, even though equiv captures an already-simplified notion of equivalence[1], a decision procedure for it does not exist, due to Rice's theorem.

To get decidability back, we define a new class of relations $\mathsf{equiv}_S \subseteq P \times P$ (whose complement is $\not{\mathsf{equiv}}_S$). We say that P_1 equiv_S P_2 (read "P_1 is S-equivalent to P_2") if, for all inputs, one of the following holds:

- Either P_1 or P_2 does not stop within S steps (we can think of each CPU cycle as one step).
- Both P_1 and P_2 fail.
- Both P_1 and P_2 stop with the same output.

The equiv_S relation captures the notion that we cannot tell P_1 and P_2 apart by running them for at most S steps. Figure 2 shows an example of two programs that are S-equivalent for $S = 10$, but not equivalent. This is the case because P_2 loops while the counter is less than 42, so with 10 steps in our "budget" we will have to stop P_2 before the loop is over and we can observe the different result.

[1] For example, equiv has a very narrow notion of output that excludes side effects.

```
# P_1
add $3, $1, $2
```

```
# P_2
  add $4, $0, 1  # counter
  add $5, $0, 42 # upper bound
loop:
  slt $6, $4, $5
  beq $6, $0, end
  add $4, $4, 1
  beq $0, $0, loop
end:
  add $3, $1, $1
```

Fig. 2. P_1 equiv_{10} P_2, but P_1 ~~equiv~~ P_2

.

Given a fixed S, the equiv_S relation is decidable because there is a finite number of inputs to try, and for each input we only need to run the programs a finite number of steps.

We already saw that equivalence not always implies S-equivalence. However, the converse always holds. The following lemma shows that equiv_S over-approximates equiv.

Lemma 1. $\forall S, P_1, P_2,\ P_1\ \mathit{equiv}\ P_2 \implies P_1\ \mathit{equiv}_S\ P_2$.

Proof. Let P_1 equiv P_2. Then we have one of two cases:

- Either P_1 or P_2 (or both) do not stop within S steps. Then by definition P_1 equiv_S P_2.
- Both P_1 and P_2 stop within S steps. Then because they are equivalent, we know that they either fail with an error, or both stop with the same output. In either case, P_1 equiv_S P_2.

Corollary 1. P_1 ~~equiv_S~~ $P_2 \implies P_1$ ~~equiv~~ P_2.

Proof. This is just the contrapositive of Lemma 1.

Corollary 1 can be used to argue the soundness (with respect to ~~equiv~~) of any decision procedure that under-approximates ~~equiv_S~~. In the next section we will show one such under-approximation based on concolic execution.

3 Concolic Disequivalence

We know from Corollary 1 that any relation that under-approximates ~~equiv_S~~ is sound. Figure 3 shows why want an under-approximation: efficiency. ~~equiv~~ captures the class of programs that are disequivalent, but is undecidable. ~~equiv_S~~ is decidable, but likely cannot be computed efficiently. Therefore, we look for a subset of ~~equiv_S~~ (an under-approximation) that can be efficiently computed.

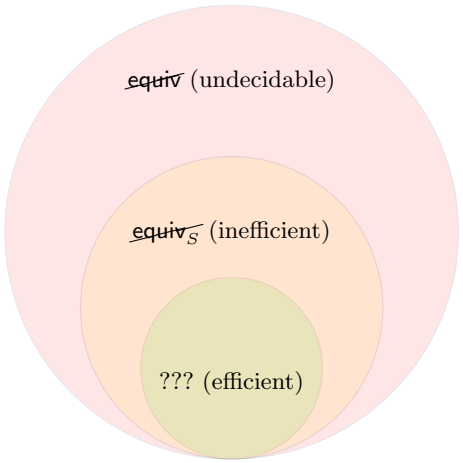


Fig. 3. Hierarchy of disequivalence relations

To fill the missing relation in Figure 3 we propose concolic disequivalence. Abstractly, concolic disequivalence is a function compare(P_1, P_2, S) that takes as inputs two MIPS programs and returns one of two answers:

- "disequivalent", in which case P_1 ~~equiv~~$_S$ P_2.
- "possibly equivalent", meaning that P_1 and P_2 might or might not be S-equivalent.

Figure 3 shows pseudocode for compare. The algorithm alternately executes P_1 and P_2. At every step, one of the programs is labelled as the "driver" and the other one as the "verifier". The driver program is then concolically executed, yielding a set of inputs that exercise a new program path (of the driver). The inputs can then be fed to the verifier, and the results of both driver and verifier compared. If the results are different, then we know P_1 and P_2 are disequivalent. Otherwise, the driver becomes the verifier, and vice-versa. Eventually, we will traverse all explorable paths, at which point P_1 and P_2 can be declared possibly equivalent.

We now give an example of how compare operates. Consider the sample programs below:

```
# P_1
  bne $1, 42, end
  add $3, $0, $0
end:
  add $3, $1, $2
```

```
# P_2
  add $3, $1, $2
  bne $2, 100, end
  add $3, $3, $2
end:
```

Figure 5 summarizes the state of the algorithm as it compares P_1 and P_2. First, notice how the driver and verifier roles flip between P_1 and P_2 in con-

```
function COMPARE(P1, P2, S)
    b ← true
    while either P1 or P2 has unexplored paths do
        if b then                                          ▷ Select driver and verifier
            D ← P1
            V ← P2
        else
            D ← P2
            V ← P1
        end if
        if D has unexplored paths then
            I ← new inputs that exercise an unexplored path
            R1 ← run(P1, I, S)
            R2 ← run(P2, I, S)
            if both P1 and P2 stopped then
                if both P1 and P2 stopped with an error then
                                                           ▷ do nothing
                else if either P1 or P2 stopped with an error then
                    return "disequivalent"
                else
                    if R1 ≠ R2 then
                        return "disequivalent"
                    end if
                end if
            end if
            mark the path discovered by I as explored
            b ← ¬b
        end if
    end while
    return "possibly equivalent"
end function
```

Fig. 4. Concolic disequivalence algorithm

secutive runs. Every row indicates the input values, as well as the outputs R_D and R_V of the driver and verifier, respectively. At every run, we also record the path taken by the driver. Path conditions are negated to make sure we explore new paths in every iteration. In the fourth iteration, we can see of compare finds that the input pair $\$1 = 1, \$2 = 100$ leads to different outputs in the driver and verifier. At this point, P_1 and P_2 are declared as disequivalent.

Notice that in order to uncover the different, it is necessary to concolically explore the paths in both P_1 and P_2, and not only of P_1. In Figure 5, runs 1 and 3 explore both branches of the conditional jump in P_1, but they exercise the same path in P_2. Only after we also execute P_2 do we find a counterexample to equivalence.

Run	Driver	Verifier	$1	$2	Path	R_D	R_V
1	P_1	P_2	1	1	$\$1 \neq 42$	2	2
2	P_2	P_1	1	1	$\$2 \neq 100$	2	2
3	P_1	P_2	42	1	$\$1 = 42$	2	2
4	P_2	P_1	1	100	$\$2 = 100$	201	2

Fig. 5. A sample execution of compare

4 Tamarin

Tamarin[2] is a Scala implementation of the compare algorithm from Section 3. We first give an overview of the major modules in Tamarin, shown in Figure 6, and then describe them in more detail in subsequent sections:

- Concolic is the entry point to Tamarin. It implements the top-level loop that visits unexplored paths, alternating between the two programs being compared. Concolic uses the other modules as helpers (in Figure 6, requests made by a module appear as solid lines, and responses to prior requests are shown with dashed lines).
- CPU is a MIPS emulator that is instrumented to record symbolic traces containing path conditions, which can later be negated to explore new paths.
- Trace consumes raw traces coming from CPU and transforms them in multiple ways so that they can be handed over to the Z3 solver.
- Query translates (modified) CPU traces into equivalent logical formulae. The formulae are then solved by the Z3 solver, producing new inputs that, if fed to the program under test, will lead to traversing unexplored paths.

[2] Tamarins are small-sized monkeys from Central and South America. They are related to marmosets, which are also New World monkeys, and less-importantly give name to the black-box submission and testing server in use at the University of Waterloo as of Fall 2017 [5].

– Z3 is an SMT solver developed at Microsoft [3]. We use it as black box for solving queries, over the theories of bitvectors and arrays, that result from program traces.

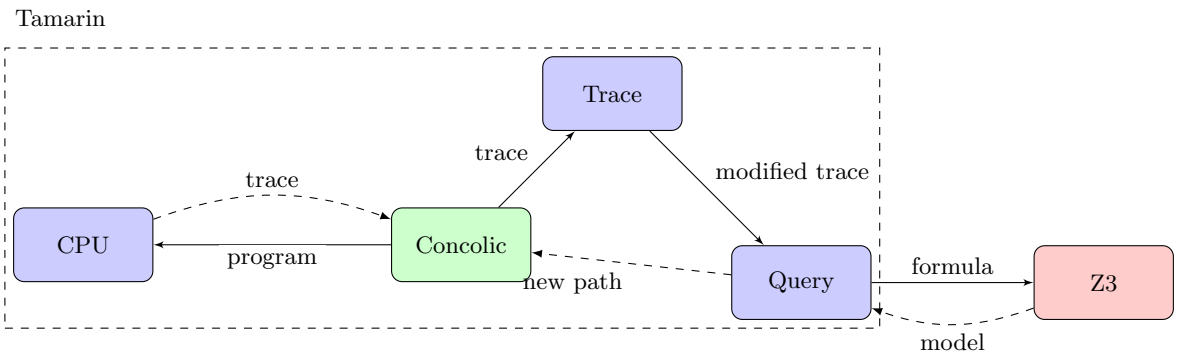


Fig. 6. Overview of Tamarin's modules

4.1 Trace Collection

The CPU module is in charge of running MIPS programs and collecting symbolic traces from executions. It is based on the MIPS emulator written by Ondřej Lhoták for CS241E at the University of Waterloo [4].

The interface to the module consists of a single function:

```
def run(prog: Seq[Word], r1: Word, r2: Word, fuel: Long): RunRes
```

The run function takes as input a program represented as as sequence of words, the values of registers $1 and $2 (the inputs to the program) and a fuel value (explained below).

The output is an algebraic data type RunRes that can take one of three forms:

```
trait RunRes
case class Done(state: State, trace: Trace) extends RunRes
case class NotDone(trace: Trace) extends RunRes
case class Error(ex: RuntimeException) extends RunRes
```

– If the program executes without error, then Done(state, trace) is returned. state is the state of the CPU after execution, including the contents of memory (which are ignored) and of register $3, the output register. trace is the symbolic trace captured during the program's execution, and is described below.

– If the program ran for more than fuel CPU cycles without stopping, then the result is NotDone(trace). Notice that even though the program did not stop we can still return a trace recording the execution right until the moment we stopped it. The fuel argument to run plays the same role as the S argument in Figure 3.
– Finally, if there was an error during program execution (for example, an attempted division-by-zero), then we return Error.

Notice that due to fuel parameter and error boxing, the augmented emulator in Tamarin, unlike a vanilla MIPS emulator, is "hardened" in the sense that it can execute MIPS programs that do not stop (the emulator itself will stop) or throw errors (will be catched at the top level by the emulator).

While the emulator is executing a program, it also records a symbolic trace of (most of) the executed instructions. We support a subset of the MIPS instruction set, containing 18 instruction types [1]. Notably, unlike in full MIPS, there are no system calls in our supported subset.

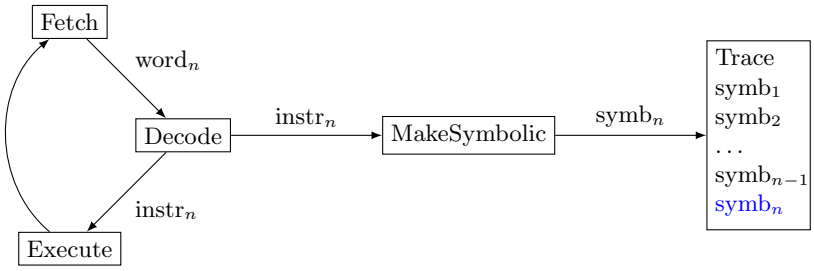


Fig. 7. Instrumented CPU that records symbolic traces

The fetch-decode-execute cycle in our emulator is modified to record a symbolic representation of each instruction as it is executed (Figure 7). The symbolic instructions are then stored in a trace. There are two types of instructions in a trace: assignments and path conditions.

– Assignments are instructions that mutate the CPU state, but do not affect the control flow. The symbolic form of most assignments is very similar to the concrete instruction that is executed. For example, the symbolic representation of the instruction `add $3, $1, $2` is $r_3 \leftarrow r_1 + r_2$.
– Path conditions are instructions that modify the control flow of the program. This set potentially contains both conditional and unconditional jumps, but we track only conditional ones. Specifically, we record the branch taken at each conditional jump, and symbolically record it as an (in)equality. For example, if the branch `beq $1, $2, foolabel` is not taken, then we will add $r_1 \neq r_2$ to our trace.

One additional point of note: the program counter (pc) is not symbolically tracked. This means Tamarin cannot reason about unconditional jumps, so only a

single path is explored for the program in Figure 8. There, Tamarin will execute P with the initial inputs (a hard-coded constant) and error out, because the jump will lead to an invalid instruction. Tamarin will miss the fact that $1 and $2 can be given values such that we could jump to either of the labels, leading to a successful execution.

```
# P
  add $4, $1, $2
  jr $4
good:
  add $3, $1, $2
  jr $31 # $31 contains the termination pc
bad:
  add $3, $1, $1
  jr $31
```

Fig. 8. Not tracking the pc leads to under-approximations

Since the pc is not tracked, whenever it is used in another instruction we need to substitute it by its concrete value. This technique is called concretization [2]. For example, the load immediate and skip instruction has the following semantics:

`lis $d` $\quad d \leftarrow Mem[pc]; pc \leftarrow pc + 4$

Instead of the (precise) interpretation above, we use concretization to generate the (under-approximating) symbolic instruction $d \leftarrow v$, where v is the value at the address following the `lis` instruction.

4.2 Transformations

The Trace module also exposes just a single function:

```
def transform(trace: Trace, depth: Int): Trace
```

transform takes as input a trace that was produced by CPU and a depth parameter (explained below), and modifies the input trace so that it can be later converted into a logical formula. The depth parameter limits the number of path conditions in the outputted trace: this is so we can bound the depth of our DFS as we explore new paths (see Section 4.4 for more details). In effect, we can think of traces as the intermediate representation (IR) for queries to the SMT solver. The Transform module can be then thought of as lowering the semantic complexity of traces.

Internally, Transform consists of multiple "mini-phases", each of type `Trace => Trace`, which are sequentially applied to the input. The mini-phases are described below.

Desugaring

Simplification

Trimming

SSA conversion

4.3 Queries

Memory, jumps, arithmetic operators.

4.4 Concolic Execution Redux

Alternation. Compatibility. Soundness/Completeness. Efficiency.

5 Evaluation

6 Related Work

7 Conclusions

References

1. MIPS reference sheet. `https://www.student.cs.uwaterloo.ca/~cs241/mips/mipsref.pdf`.
2. Robin David, Sébastien Bardin, Josselin Feist, Laurent Mounier, Marie-Laure Potet, Thanh Dinh Ta, and Jean-Yves Marion. Specification of concretization and symbolization policies in symbolic execution. In *ISSTA*, pages 36–46, 2016.
3. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.
4. Ondrej Lhotak. CS241e - foundations of sequential programs (enriched). `https://www.student.cs.uwaterloo.ca/~cs241e/`.
5. Jaime Spacco, William Pugh, Nat Ayewah, and David Hovemeyer. The marmoset project: an automated snapshot, submission, and testing system. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 669–670. ACM, 2006.