Hobbit: A Tool for Contextual Equivalence Checking Using Bisimulation Up-to Techniques

Vasileios Koutavas[1⋆], Yu-Yang Lin[1⋆], and Nikos Tzevelekos[2]

[1] Trinity College Dublin, Ireland {Vasileios.Koutavas,linhouy}@tcd.ie
[2] Queen Mary University of London, UK nikos.tzevelekos@qmul.ac.uk

Abstract. We present a bounded equivalence verification tool called Hobbit for higher-order programs with local state—based on a subset of OCaml—that combines fully abstract *symbolic environmental bisimulations*, novel *up-to techniques*, and lightweight *invariant annotations*. The tool reports no false positives or negatives, can automatically detect all inequivalences, and is able to automatically or semi-automatically prove many equivalences, including all classical Meyer and Sieber equivalences.

Keywords: Contextual equivalence · bounded model checking · symbolic bisimulation · up-to techniques · operational game semantics.

1 Introduction

Hobbit[3] is a prototype tool implemented in OCaml that performs bounded model-checking for contextual equivalence. It can fully-automatically detect inequivalent program expressions and prove hard equivalences automatically or semi-automatically. It implements a novel technique based on *symbolic environmental bisimulations* and *up-to techniques* for the entirety of a higher-order language with local state based on a subset of OCaml. For our setting, contextual equivalence is a relation over program expressions which guarantees that related expressions are interchangeable in any program context. A technical report describing this technique in detail can be found in [4].

Bounded model checking [1], while proficient in bug finding [2,3,7], can rarely prove the absence of errors: a bound is usually reached before all possible program runs (potentially infinite in number) are explored. We propose two technologies inspired by hand-written bisimulation proofs to significantly increase the number of equivalences proven by Hobbit, including for example all classical equivalences due to Meyer and Sieber [6]. Firstly, we use up-to techniques to finitise infinite transition systems, including examples with divergence. This turns Hobbit from a bounded checker into an equivalence prover. Secondly, with *state invariant* annotations, also based on novel up-to techniques, we allow the user to abstract over concrete stored values and replace them with symbolic values satisfying

⋆ Koutavas & Lin were supported by Science Foundation Ireland grant 13/RC/2094 (Lero)

[3] Hobbit stands for Higher Order Bounded BIsimulation Tool, and is available at https://github.com/LaifsV1/Hobbit.

predicates. This allows our analysis to reach a fixpoint where HOBBIT cannot automatically handle a store that changes indefinitely.

Due to the undecidable nature of contextual equivalence, no set of up-to techniques is guaranteed to finitise all examples. However, our experience with HOBBIT shows that the addition of a small number of up-to techniques can turn a bounded checker into a powerful equivalence verification tool capable of handling all classical example equivalences (e.g. all in [6]) with the exception of those with synchronising internal loops and those based on stack properties.

2 Up-to Techniques

We make use of two novel up-to techniques, *up to separation* and *up to re-entry*, to deal with infinity in the LTS due to higher-order interactions with the context, and two more, *up to abstraction* and *up to tautology*, to enable the use of state invariants in equivalence proofs.

2.1 Up to Separation

The intuition of this technique is that if different functions operate on disjoint parts of the store, they can be explored in disjoint parts of the bisimulation transition system. Taken to the extreme, it suffices to apply a function that does not contain free locations only once in a bisimulation test as two copies of the same function cannot interfere with each other, even if they allocate new locations after the application. Our experience with HOBBIT has shown this is one of the most effective techniques at finitising the analysis.

Example 1. The following is a classic example equivalence from Meyer and Sieber [6]. The following terms are equivalent at type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$.

$M =$ `fun f -> ref x = 0 in f ()` $\qquad$ $N =$ `fun f -> f ()`

Here, the context is able to grow the evaluation stack indefinitely by reentering M and N, thus producing an infinite LTS. Applying up-to separation immediately after the first call, since M and N do not contain free locations, a second call to the functions is prevented, which results in a trivially small LTS. □

2.2 Up to Proponent Function Re-entry

The intuition here is that if calling related functions cannot change the local stores (up to garbage collection) or increase the knowledge of the context, then there are no additional observations to be made by nested calls to said functions. This has similarities to environmental bisimulation with induction hypotheses [5].

In HOBBIT we require the user to flag the functions where the up to re-entry technique should be applied. This annotation is later combined with state invariant annotations, as they are often used together.

Example 2. The following equivalence relates Landin's imperative fixpoint operator with a fixpoint with letrec. The type is $((\mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}) \to \mathsf{int} \to \mathsf{int}$.

$M =$
```
let landinsfixpoint f =
  ref x = fun z -> z in
  x:= (fun y {} -> f !x y); !x
in landinsfixpoint
```

$N =$
```
let rec fix f =
  (fun y -> f (fix f) y)
in fix
```

In this example, up to separation prevents the outer functions from being applied more than once. However the inner functions `(fun y {} -> f !x y)` and `(fun y -> f (fix f) y)`, which are provided as arguments to context function `f`, cannot be eliminated by up to separation because they can access location `x`. Up to re-entry removes the need to consider nested calls to these functions. The syntax `{}` tells Hobbit where to apply this technique. □

3 State Invariants: Up to Abstraction and Tautology

Up to abstraction allows us to abstract constants with fresh symbolic values and instead prove equivalent the more abstract configurations. Up to tautology allows us to introduce tautologies into the symbolic environments. Combining the two techniques we can introduce invariants about values in the store.

Currently in Hobbit, up to abstraction and tautology are combined and applied in a principled way. Functions can be annotated with the following syntax:

F `= fun x {`$\vec{\kappa}$ `|` l_1 `as` $C_1[\vec{\kappa}]$`, ...,` l_n `as` $C_n[\vec{\kappa}]$ `|` ϕ`} -> e`

Example 3. Using up to abstraction and tautology, Hobbit is able to prove the example below, an adaptation from [5]. The invariant relates locations in the two terms by assigning the same $\vec{\kappa}$ to `wp`, `w1` and `w2` in both programs.

```
M = ref y = 0 in
    let set z = {wp, wy1, wy2, wy | y as wy |
       ((wp mod 2 = 0) && (wy = wy1))
       || ((wp mod 2 <> 0) && (wy == wy2))} -> y := z in
    let get () = !y in
    (set , get)

N = ref y1 = 0 in ref y2 = 0 in ref p = 0 in
    let set1 z =
      {wp, wy1, wy2, wy | p as wp; y1 as wy1; y2 as wy2 | true}
      -> p := !p + 1; if !p mod 2 = 0 then y1 := z else y2 := z in
    let get1 () = if !p mod 2 = 0 then !y1 else !y2 in
    (set1 , get1)
```

4 Implementation and Evaluation

We ran Hobbit on a test-suite of 108 equivalences and 68 inequivalences—3894 and 2263 lines of code for equivalences and inequivalences respectively. Examples were checked in five configurations: default (all up-to techniques on), up to separation off, annotations (up to state abstraction, tautology, and re-entry) off, up to re-entry off, and everything off. The tool stops either on the first trace that disproves equivalence, after enumerating all traces up to the bound, or after timing out at 150 seconds. Time taken and exit status were recorded for each example; an overview of the experiment can be seen in the following table. All experiments ran on an Ubuntu 18.04 machine with 32GB RAM, Intel Core i7 1.90GHz CPU, with intermediate calls to Z3 4.8.10 to prune invalid internal symbolic branching and decide symbolic bisimulation conditions.

	default	sep. off	annot. off	ree. off	all off
eq.	75/0 (6.3s)	32/0 (1665.5s)	50/0 (178.9s)	60/0 (177.9s)	3/0 (2274.2s)
ineq.	0/68 (20.0s)	0/66 (312.8s)	0/68 (19.6s)	0/68 (20.1s)	0/65 (515.7s)
a/b (c) for a equivalences and b inequivalences found taking c seconds in total					

References

1. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. Springer Berlin Heidelberg (1999)
2. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Springer Berlin Heidelberg (2004)
3. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for java bytecode. In: TACAS. Springer (2019)
4. Koutavas, V., Lin, Y., Tzevelekos, N.: There and back again: From bounded checking to verification of program equivalence via symbolic up-to techniques. CoRR **abs/2105.02541** (2021), https://arxiv.org/abs/2105.02541
5. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: POPL. ACM (2006)
6. Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: POPL. Association for Computing Machinery (1988)
7. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: FMICS (2015)