The WhyRel Prototype for Modular Relational Verification of Pointer Programs

Ramana Nagasamudram[1]([✉]), Anindya Banerjee[2], and David A. Naumann[1]

[1] Stevens Institute of Technology, Hoboken, USA
{rnagasam,dnaumann}@stevens.edu
[2] IMDEA Software Institute, Madrid, Spain
anindya.banerjee@imdea.org

Abstract. Verifying relations between programs arises as a task in various verification contexts such as optimizing transformations, relating new versions of programs with older versions (regression verification), and noninterference. However, relational verification for programs acting on dynamically allocated mutable state is not well supported by existing tools, which provide a high level of automation at the cost of restricting the programs considered. Auto-active tools, on the other hand, require more user interaction but enable verification of a broader class of programs. This article presents WhyRel, a tool for the auto-active verification of relational properties of pointer programs based on relational region logic. WhyRel is evaluated through verification case studies, relying on SMT solvers orchestrated by the Why3 platform on which it builds. Case studies include establishing representation independence of ADTs, showing noninterference, and challenge problems from recent literature.

Keywords: local reasoning · relational verification · auto-active verification · data abstraction.

AQ1

1 Introduction

Relational properties encompass conditional equivalence of programs (as in regression verification [28]), noninterference (in which a program is related to itself via a low-indistinguishability relation), and other requirements such as sensitivity [6]. The problem we address concerns tooling for the modular verification of relational properties of heap-manipulating programs, including programs that act on differing data representations involving dynamically allocated pointer structures.

Modular reasoning about pointer programs is enabled through local reasoning using frame conditions, procedural abstraction (i.e., reasoning under hypotheses about procedures a program invokes), and data abstraction, requiring state-based encapsulation. For establishing properties of ADTs such as representation independence, encapsulation plays a crucial role, permitting implementations to rely on invariants about private state hidden from clients. Relational verification also involves a kind of compositionality, the *alignment* of intermediate execution steps, which enables use of simpler relational invariants and specs (see e.g. [29,17,25]).

We aim for auto-active verification [19], accessible to developers, as promoted by tools such as Dafny and Why3. Users are expected to provide specifications, annotations such as loop invariants and assertions, and, for relational verification, alignment hints. The idea is to minimize or eliminate the need for users to manually invoke tactics for proof search.

Automated inference of specs, loop invariants, or program alignments facilitates automated verification, and is implemented in some tools. But in the current state of the art these techniques are restricted to specs and invariants of limited forms (e.g., only linear arithmetic) and seldom support dynamically allocated objects. So inference is beyond the scope of this paper.

What is in scope is use of strong encapsulation, to hide information in the sense that method specs used by clients do not expose internal representation details, and to enable verification of modular correctness of a client, in the sense that its behavior is independent from internal representations. Achieving strong encapsulation for pointer programs, without undue restriction on data and control structure, is technically challenging. Auto-active tools rely on extensive axiomatization for the generation of *verification conditions (VCs)*; for high assurance the VCs should be justified with respect to a definitional operational semantics of programs and specs.

In this article, we describe WhyRel, a prototype for auto-active verification of relational properties of pointer programs. Source programs are written in an imperative language with support for shared mutable objects (but no subtyping), dynamic allocation, and encapsulation. The assertion language is first-order and, for expressing relational properties, includes constructs that relate values of variables and pointer structures between two programs. WhyRel is based on relational region logic [1], a relational extension of region logic [4,2]. Region logic provides a flexible approach to local reasoning through the use of *dynamic frame* conditions [15] which capture *footprint*s of commands acting on the heap. Verification involves reasoning explicitly about regions of memory and changes to them as computation proceeds; flexibility comes from being able to express notions such as parthood and separation in the same first-order setting.

Encapsulation is specified using a kind of dynamic frame, called a *dynamic boundary*: a footprint that captures a module's internal locations. Enforcing encapsulation is then a matter of ensuring that clients don't directly modify or update locations in a module's boundary. There are detailed soundness proofs for the relational logic [1], of which our prototype is a faithful implementation.

WhyRel is built on top of the Why3 platform[3] for deductive program verification which provides infrastructure for verifying programs written in WhyML, a subset of ML [7] with support for ghost code and nondeterministic choice. The assertion language is a polymorphic first-order logic extended with support for algebraic data types and recursively and inductively defined predicates [11]. Why3 generates VCs for WhyML which can then be discharged using a wide array of theorem provers, from interactive proof assistants such as Coq and Isabelle, to first-order theorem provers and SMT solvers such as Vampire, Alt-Ergo and Z3.

---

[3] The Why3 distribution can be found at: https://why3.lri.fr/.

Primarily, WhyRel is used as a front end to Why3. Users provide programs, specs, annotations, and for relational verification, relational specs and alignment specified using a specialized syntax for product programs. WhyRel translates source programs into WhyML, performing significant encoding so as to faithfully capture the heap model and fine-grained framing formalized in relational region logic. VCs pertinent to this logic are introduced as intermediate assertions and lemmas for the user to establish. Verification is done using facilities provided by Why3 and the primary mode of interaction is through an IDE for viewing and discharging verification conditions.

Our approach is evaluated through a number of case studies performed in WhyRel, for which we rely entirely on SMT solvers to discharge proof obligations. The primary contribution is the development of a tool for relational verification of heap manipulating programs which has been applied to challenging case studies. Examples formalized demonstrate the effectiveness of relational region logic for alignment, for expressing heap relations, and for relational reasoning that exploits encapsulation.

Organization. Sec. 2 highlights aspects of specifying programs and relational properties in WhyRel using a stack ADT example. Sec. 3 discusses examples of program alignment. Sec. 4 gives an overview of the design of WhyRel and Sec. 5 provides highlights on experience using the tool. Sec. 6 discusses related work and Sec. 7 concludes.

2   A tour of WhyRel

Programs and specifications. WhyRel provides a lightweight module system to organize definitions, programs, and specs. Developments are structured into interfaces and modules that implement interfaces. In addition, for relational verification, WhyRel introduces the notion of a *bimodule*, described later, to relate method implementations between two (*unary*) modules.

We'll walk through aspects of specification in WhyRel using the `STACK` interface shown in Fig. 1, which describes a stack of boxed integers with push and pop operations. The interface starts by declaring global variables, `pool` and `capacity`, and client-visible fields of the `Cell` and `Stack` classes. Variable `pool` has type `rgn`, where a *region* is a set of references, and is used to describe objects notionally owned by modules implementing the stack interface; `capacity` has type `int` and describes an upper bound on the size of a stack. The `Cell` class for boxed integers is declared with a single field, `val`, storing an `int`. The `Stack` class is declared with three fields: `rep` of type region keeps track of objects used to represent the stack, `size` of type `int` stores the number of elements in the stack, and the ghost field `abs` of type `intlist` (list of mathematical integers) keeps track of an abstraction of the stack, used in specs. Class definitions can be refined later by modules implementing the interface: e.g., a module using a linked-list implementation might extend the `Stack` class with a field `head` storing a reference to the list.

Heap encapsulation is supported at the granularity of modules through the use of *dynamic module boundaries* which describe locations internal to a module. A

```
interface STACK =
  public pool:rgn /* rgn: a set of references */  public capacity:int
  class Cell {val:int}  class Stack {rep:rgn; size:int; ghost abs:intlist}

  /* encapsulated locations */
  boundary {capacity, pool, pool‘any, pool‘rep‘any}

  public invariant stkPub = ∀ s: Stack ∈ pool. 0 ≤ s.size ≤ capacity
    ∧ (∀ t: Stack ∈ pool. s ≠ t ⇒ s.rep ∩ t.rep ⊆ {null}) ∧ ...

  meth Cell(self: Cell) : unit ...    meth getVal(self: Cell) : int ...
  meth Stack(self: Stack) : unit ensures {self ∈ pool} ...

  meth push(self: Stack, k: int) : unit
    requires {self ∈ pool ∧ self.size < capacity}
    ensures  {self.abs = cons(k,old(self.abs)) ∧ ...}
    /* allowed heap effects of implementations */
    effects  {rw {self}‘any, self.rep‘any, alloc; rd self,capacity}

  meth pop(self: Stack) : Cell
    requires {self ∈ pool ∧ self.size > 0}
    ensures  {self.size = old(self.size)-1}
    ensures  {result.val = hd(self.abs) ∧ self.abs = tl(old(self.abs))}
```

Fig. 1: WhyRel interface for the Stack ADT

location is either a variable or a heap location $o.f$, where o is an object reference and f is its field. In WhyRel, module boundaries are specified in interfaces and clients are enforced to not directly read or write locations described by the boundary except through the use of module methods. For our stack example, the dynamic boundary is `capacity, pool, pool‘any, pool‘rep‘any`; expressed using image expressions and the `any` datagroup. Given a region G and a field f of class type, the *image expression* $G‘f$ denotes the region containing the locations $o.f$ of all non-null references o in G, where f is a valid field of o. If f is of type region, $G‘f$ is the union of the collection of reference sets $o.f$ for all o in G. For f of primitive type, such as `int` or `intlist`, $G‘f$ is the empty region. The *datagroup* `any` is used to abstract from concrete field names: the expression `pool‘any` is syntactic sugar for `pool‘val,...,pool‘abs`. Intuitively, the dynamic boundary in Fig. 1 says that clients may not directly read or write `capacity`, `pool`, any fields of objects in `pool`, and any fields of objects in the `rep` of any `Stack` in `pool`.

While encapsulation is specified at the level of modules, separation or locality at finer granularities can be specified using module invariants. The stack interface defines a public invariant `stkPub` which asserts that the `rep` fields of all `Stack` objects in `pool` are disjoint. This idiom can be used to ensure that modifying one object has no effect on any locations in the representation of another. Clients can rely on public invariants during verification, but modules implementing the interface must ensure they are preserved by module methods. Additionally, modules may define private invariants that capture conditions on internal state; provided these refer only to encapsulated locations, i.e., the designated boundary *frames* these invariants, clients are exempt from reasoning about them [14].

```
module Client =
  meth prog (n: int) : int
    requires { 0 ≤ n < capacity ∧ ... }
    effects  { rw alloc, pool, pool`any, pool`rep`any; rd n, capacity }
  = var i: int in var c: Cell in
    var stk: Stack in stk := new Stack; Stack(stk);
    while (i < n) do push(stk,i); i:=i+1 done; i := 0;
    while (i < n) do c:=pop(stk); result:=result+getVal(c); i:=i+1 done;

  meth prog (n: int|n: int) : (int|int)   /* Relational spec for prog */
    requires { n =̇ n ∧ Both(0 ≤ n < capacity ∧ ... ) }
    ensures  { result =̇ result }
```

Fig. 2: Example client for STACK and relational spec for equivalence

Finally, the STACK interface defines specs for initializers (methods Cell and Stack) and public specs for client-visible methods getVal, push, and pop. Notice that the stack initializer ensures self is added to the boundary (through post self ∈ pool) and stack operations require self to be part of the boundary (through pre self ∈ pool). Specs for push and pop are standard, using "old" expressions to precisely capture field updates. WhyRel's assertion language is first-order and includes constructs such as the points-to assertion $x.f = e$ and operations on regions such as subset and membership. In addition to pre- and post-conditions, each method is annotated with a frame condition in an effects clause that serves to constrain heap effects of implementations. Allowable effects are expressed using read/write (rw) or read (rd) of locations or location sets, described by regions. For example, the effects clause for push says that implementations may read/write any field of self and any field of any objects in self.rep. The distinguished variable alloc is used to indicate that push may dynamically allocate objects.

In our development, we build two modules that implement the interface in Fig. 1: one using arrays, ArrayStack and another using linked-lists, ListStack. Both rely on private invariants on encapsulated state that capture constraints on their pointer representations and its relation to abs, the mathematical abstraction of stack objects. The private invariant of ListStack, for example, says that Cell values in the linked-list of any Stack in pool are in correspondence with values stored in abs.

Example client, equivalence spec, and verification. We now turn attention to an example client, prog, shown in Fig. 2. This program computes the sum $\Sigma_{i=0}^{n}i$, albeit in a roundabout fashion, using a stack. The frame condition of prog mentions the boundary for STACK, but this is fine since the client respects WhyRel's encapsulation discipline, modifying encapsulated locations solely through calls to methods declared in the STACK interface. For this client, our goal is to establish equivalence when linked against either implementations of STACK. Let the *left* program be the client linked against ArrayStack, and the *right* the client linked against ListStackEquivalence is expressed using the relational spec shown in Fig. 2. For brevity, we omit frame conditions when describing relational specs.

```
meth prog (n: int | n: int) : (int | int)
= var i: int | i: int in var c: Cell | c: Cell in
  var stk: Stack in ⌊ stk := new Stack ⌋; ⌊ Stack(stk) ⌋;
  while (i < n) | (i < n) do ⌊ push(stk,i) ⌋; ⌊ i:=i+1 ⌋ done; ⌊ i:=0 ⌋;
  while (i < n) | (i < n) do ⌊ c:=pop(stk) ⌋;
    ⌊ result:=result+getVal(c) ⌋; ⌊ i:=i+1 ⌋ done;
```

Fig. 3: Alignment for example stack client

This relational spec relates two versions of `prog`; the notation `(n:int | n:int)` is used to declare that both versions expect `n` as argument. The pre-relation requires equality of inputs: $n \doteq n$ says that the value of `n` on the left is equal to the value of `n` on the right. We use $(\doteq)$, instead of $(=)$ to distinguish between values on the left and the right[4]. The relational spec requires the two states being related to satisfy the unary precondition for the client, as indicated by `Both(...)`. The post-relation, `result` $\doteq$ `result`, asserts equality on returned values. In WhyRel, relational specs capture a ∀∀ termination-insensitive property: *terminating executions of the programs being related, when started in states related by the pre-relation, will result in states related by the post-relation.*

WhyRel supports two approaches to verifying relational properties. The first reduces to proving functional properties of the programs involved. For instance, equivalence of the client when linked against the two stack implementations is immediate if we prove that `prog` indeed computes the sum of the first `n` nonnegative integers.

However, this approach neither lends well to more complicated programs and relational properties, nor does it allow us to exploit similarities between related programs or reason modularly using relational specs. The alternative is to prove the relational property using a convenient alignment of the two programs. Alignments are represented syntactically in WhyRel using *biprograms* which pair points of interest between two programs so that their effects can be reasoned about in tandem. If the chosen alignment is *adequate* in the sense of capturing all pairs of executions of the related programs, relational properties of the alignment entail the corresponding relation between the underlying programs.

The biprogram for `prog` is shown in Fig. 3. The alignment it captures is maximal: every control point in one version of the client is paired with itself in the other version. The construct $(C|C')$ pairs a command C on the left with a command C' on the right, and the *sync* form $\lfloor C \rfloor$ is syntactic sugar for $(C|C)$; e.g., the biprogram for `prog` aligns the two allocations using $\lfloor$`stk := new Stack`$\rfloor$. Further, this biprogram aligns both loops in *lockstep*, indicated using the syntax `while e|e' do ... done`. This alignment pairs a loop iteration on the left with a loop iteration on the right and requires the loop guards be in agreement: here, that `i < n` on the left is true just when `i < n` on the right is. Calls to stack operations are aligned in the loop body using the sync construct to facilitate

---

[4] Note in particular that $x \doteq y$ is not the same as $y \doteq x$

```
bimodule REL_STACK (ArrayStack | ListStack) =
  coupling stackCoupling = ∀ s: Stack ∈ pool | s: Stack ∈ pool.
    s ≐ s ⇒ s.abs ≐ s.abs ∧ ...
  meth Stack(self: Stack | self: Stack) : (unit | unit)
    ensures {self ≐ self ∧ ...} = /* biprogram for Stack */
  meth push(self: Stack | self: Stack) : (unit | unit)
    requires {self ≐ self ∧ ... }
    ensures  {self.abs ≐ self.abs ∧ ... } = /* biprogram for push */
  meth pop(self:Stack | self:Stack) : (Cell | Cell)
    requires {self ≐ self ∧ Both (self ∈ pool) ∧ Both (self.size > 0)}
    ensures  {... ∧ result.val ≐ result.val} = /* biprogram for pop */
```

Fig. 4: Bimodule for Stack; excerpts

modular verification of relational properties by indicating that relational specs for push and pop are to be used.

To prove the spec (in Fig. 2) about the biprogram in Fig. 3 we reason as follows: after allocation stk on both sides is initialized to be the empty stack. The first lockstep aligned loop which pushes integers from $0, \ldots, n$ maintains as invariant equality on i and on the mathematical abstractions the two stacks represent, i.e., i ≐ i ∧ stk.abs ≐ stk.abs. The second lockstep aligned loop which pops the stacks and increments result maintains as invariant agreement on the stack abstractions and result, the key conjunct being result ≐ result. This is sufficient to establish the desired post-relation. Importantly, the loop invariants are simple to prove—they only contain equalities between variables—and we don't have to reason about the exact contents of the two stacks involved.

Relational specs for Stack and verification. The reasoning described above relies on knowing the method implementations in ArrayStack and ListStack are equivalent. We need relational specs for push which state that given related inputs, the contents represented by the two stacks are the same; and for pop, which state that given related inputs, the values of the returned Cells are the same.

Fig. 4 shows a bimodule, REL_STACK, relating the two implementations of STACK. It includes relational specs for the stack operations along with biprograms used for verification. The bimodule maintains a *coupling* relation which relates data representations used by the two stack implementations. Concretely, the coupling here states that related stacks in pool represent the same abstraction. Note that quantifiers in relation formulas bind pairs of variables; and the equality s ≐ s in stackCoupling is not strict pointer equality, but indicates correspondence. Strict pointer equality is too strong as it would not allow for modeling allocation as a nondeterministic operation or permit differing allocation patterns between programs being related. Behind the scenes, WhyRel maintains a partial bijection π between allocated references in the two states being related. The relation x ≐ y, where x and y are pointers, states that x in the left state is in correspondence with y in the right state w.r.t π, i.e., $\pi(x) = y$.

The relational spec for the initializer Stack ensures self ≐ self, which is required in the specs for push and pop. Like other invariants, coupling relations

```
meth mult(n: int, m: int) =          meth mult(n:int, m:int) =
  i := 0;                              i := 0;
  while (i < n) do j:=0;                while (i < n) do
    while (j < m) do                     result := result+m;
      result := result+1; j := j+1       i := i+1
    done; i := i+1 done;               done;
```

Fig. 5: Two versions of a simple multiplication routine

are meant to be framed by the boundary and are required to be preserved by module methods being related. Encapsulation allows for coupling relations to be hidden so that clients are exempt from reasoning about them.

The steps taken to complete the Stack development and verify equivalence of two versions of its client are as follows: (i) build the STACK interface in WhyRel, with public invariants clients can rely on and a boundary that designates encapsulated locations; (ii) develop two modules refining this interface, ArrayStack and ListStack, and verify that their implementations conform to STACK interface specs, relying on any private invariants that capture conditions on encapsulated state; (iii) provide a bimodule relating the two stack modules and prove equivalence of stack operations, relying on a coupling relation that captures relationships between pointer structures used by the two modules; (iv) verify the client with respect to specs given in STACK and prove it respects WhyRel's encapsulation regime; and finally (v) develop a bimodule for the client and verify equivalence using relational specs for stack methods.

3   Patterns of alignment

Well chosen alignments help decompose relational verification, allowing for the use of simple relational assertions and loop invariants. In this section, we'll look at examples of biprograms that capture alignments that aren't maximal, unlike the STACK client example in Sec. 2. We don't formalize the syntax of biprograms here, but we show representative examples. When discussing examples, we'll omit frame conditions and other aspects orthogonal to alignment.

Differing control structures. Churchill et al. [8] develop a technique for proving equivalence of programs using state-dependent alignments of program traces. They identify a challenging problem for equivalence checking, shown in Fig. 5, which compares two procedures for multiplication with different control flow. For automated approaches to relational verification, their example is challenging because of the need to align an unbounded number m of loop iterations on the left with a single iteration on the right.

To prove equivalence, we verify the biprogram shown in Fig. 6 with respect to a relational spec with pre-relation $n \stackrel{.}{=} n \wedge m \stackrel{.}{=} m$ and post-relation result $\stackrel{.}{=}$ result; i.e., agreement on inputs results in agreement of outputs. Unlike the stack client biprogram shown in Fig. 3, the alignment embodied here is not maximal—indeed, such alignment would not be possible due to the differing

```
meth mult(n: int, m: int | n: int, m: int) : (int | int) =
  ⌊ i := 0 ⌋;
  while (i < n) | (i < n) do  invariant { i ≐ i ∧ result ≐ result }
    ( j := 0; while (j < m) do result := result+1; j := j+1 done
    | result := result+m );
    assert { ⦇result = old(result)+m⦈ };
    ⌊ i := i+1 ⌋ done;
```

Fig. 6: Biprogram for example in Fig. 5

```
meth sumpub (l: List) : int =        meth sumpub (l: List | l: List) : int =
  p:=l.head; s:=0;                     ⌊ p:=l.head ⌋; ⌊ s:=0 ⌋;
  while (p ≠ null) do                  while (p ≠ null) | (p ≠ null) .
    if p.pub then                        ⦇ ¬ p.pub ⦈ | ⦈ ¬ p.pub ⦉ do
      s:=s+p.val                       ( if p.pub then s:=s+p.val end;
    end;                                 p:=p.nxt
    p:=p.nxt                           | if p.pub then s:=s+p.val end;
  done;                                  p:=p.nxt)
  result:=s;                          done; ⌊ result:=s ⌋;
```

Fig. 7: Summing up public elements of a linked list: program and alignment

control structure. Similarities are still exploited by aligning the outer loops in lockstep and the left inner loop with the assignment to `result` on the right.

A simple relational loop invariant which asserts agreement on `i` and `result` is sufficient for proving equivalence. To show this is invariant, we need to establish that the inner loop on the left has the effect of incrementing `result` by `m`, thereby maintaining equality on `result` after the inner loop. In Fig. 6 this is indicated by the assertion after the left inner loop. The notation $⦇P⦈$ (resp. $⦉P⦉$) is used to state that the unary formula P holds in the left (and resp. right) state.

Conditionally aligned loops. Examples so far have concerned lockstep aligned loops, requiring a one-to-one correspondence between loop iterations. However, this condition is often too restrictive. WhyRel provides for other patterns of loop alignment, including those that account for conditions on data values. Consider for example the program shown in Fig. 7 which traverses a linked list and computes the sum of all elements marked public, indicated in each element's `pub` field. The program satisfies the following noninterference property, with relational spec:

```
meth sumpub(l: List | l: List) : (int | int)
  requires { Both(listpub(l,xs)) ∧ xs ≐ xs }
  ensures { result ≐ result }
```

Here `listpub(l,xs)` is a predicate which asserts that the *sequence* of public values reachable from the list pointer `l` is realized in `xs`, a mathematical list of integers. Intuitively, this specification captures the property that the result of `sumpub` does not depend on the values of nonpublic elements in the input list `l`. Showing the program computes exactly the sum of public elements: `result = sum(xs)` would imply the desired noninterference property. However, to showcase support

WhyRel offers for non-lockstep alignments, we'll establish noninterference by conditionally aligning the loops in the two copies of `sumpub` (see Fig. 7).

The alignment is as follows: if `p` is a nonpublic node on one side, perform a loop iteration on that side, pausing the iteration on the other; and if `p` on both sides is public, perform lockstep iterations of both loops. This has the effect of incrementing `s` exactly when both sides are visiting public nodes, the values of which are guaranteed to be the same by the relational precondition. The biprogram expresses this alignment through the use of additional annotations, called *alignment guards* which are general relation formulas and express conditions that lead to left-only, right-only, or lockstep iterations. The left alignment guard $(\neg$ `p.pub`$\rangle$ indicates that left-only loop iterations are to be performed when `p` on the left is not public. The right alignment guard expresses a similar condition when `p` on the right is not public. Iterations proceed in lockstep when both alignment guards are false, i.e., when `Both(p.pub)` is true.

This biprogram maintains $\exists$ `xs|xs.` `Both(listpub(p,xs))` $\wedge$ `xs`$\dot=$`xs` $\wedge$ `s`$\dot=$`s` as loop invariant, which implies the desired post-relation. This invariant states that `p` on both sides points to the same sequence of public values as captured by `listpub(p,xs)` and that there is agreement on the sum `s` computed so far. During verification, we must establish that left-only, right-only, and lockstep iterations of the aligned loops preserve this invariant. Due to the alignment, the value of `s` is only updated during lockstep iterations and its straightforward to show preservation. For one-sided iterations, reasoning relies on knowing that the sequence of public values pointed to by `p` remains the same.

4   Encoding and design

We implement WhyRel in OCaml, relying on a library provided by Why3 for constructing WhyML parse trees. Source programs are parsed and typechecked before being translated to WhyML. Prior to translation, WhyRel performs a variety of checks and transformations: primary among these is a check that clients respect encapsulation and that any biprograms provided by users are adequate. Proof obligations pertinent to relational region logic are generated in the form of intermediate assertions in WhyML programs and lemmas for the user to prove. In this section, we provide an overview of some aspects of our implementation, focusing on the translation to WhyML.

Encoding program states. References are represented using an abstract WhyML type `reference` with a distinguished element, `null`. The only operation supported on reference values is equality; WhyRel does not deal with pointer-arithmetic. Regions are encoded as ghost state, using a library for mathematical sets provided by Why3. Set operations on regions are inherently supported, and we axiomatize image expressions: for each field f, WhyRel generates a Why3 function symbol `img_f` along with an axiom that captures the meaning of $G`f$.

Program states are encoded using WhyML records. An example is shown in Fig. 8. The `state` type includes at least two mutable components called `alloct`

```
/* class defs */
class Cell {
  val: int;
  ghost rep: rgn; }

class Node {
  curr: Cell;
  nxt: Node; }

/* global vars */
public pool : rgn
```

```
type reftype = Cell | Node (*class names*)
type heap = {
  mutable val: map reference int;
  mutable ghost rep : map reference Rgn.t;
  mutable curr: map reference reference;
  mutable nxt: map reference reference }
type state = {
  mutable alloct: map reference reftype;
  mutable heap: heap;
  mutable ghost pool: rgn }
invariant {¬(Map.mem null alloct) ∧ ...}

(* axiomatization of G`nxt *)
function img_nxt : state → Rgn.t → Rgn.t
axiom img_nxt_ax : ∀ s, r, p.
  Rgn.mem p (img_nxt s r) ⇔ ∃ q.
      s.alloct[q] = Node ∧ Rgn.mem q r
    ∧ p = s.head.nxt[q]
```

Fig. 8: State encoding: WhyRel source on left, encoding in WhyML on right.

and `heap`. The component `alloct` stores a map from references to object types and keeps track of allocated objects; `heap` is itself a record with one mutable component per field in the source program that stores a map from references to values. The set of values includes references, Why3 mathematical types such as arrays and lists, regions, and primitive types such as `int` and `bool`. In addition, the `state` type contains one mutable field per global variable in the source program, storing a value of the appropriate type. The `state` type is annotated with a WhyML invariant that captures well-formedness. This invariant includes conditions such as `null` never being allocated, no dangling references, and typing constraints: for example, the `nxt` field of a `Node` is itself a `Node`.

Translating unary programs and effects. WhyRel translates unary programs into WhyML functions that act on our encoding of states. Commands that modify the heap are modeled as updates to an explicit state parameter, and local variables, parameters, and the distinguished `result` variable are encoded using WhyML reference cells. Object parameters are modeled using the `reference` type and a typing assumption. Translation of control flow statements is straightforward. For programs with loops, WhyRel additionally adds a diverges clause to the generated WhyML function: this indicates that the function may potentially diverge, avoiding generation of VCs for proving termination. While Why3 supports reasoning about total correctness, we're only concerned with partial correctness. Fig. 9 shows an example translation.

Translation of frame conditions requires care given our encoding of states. As an example, the writes for method m shown in Fig. 9 would include rw {c}`val due to the write to, and read of, field `val` of object `c`. Correspondingly, in the Why3 translation, component `val` of `s.heap` is updated; so specifying the function in Why3 requires adding `writes {s.heap.val}` as annotation. However, this isn't the granularity we want since it implies the field `val` of any reference can be

```
meth m (c: Cell, i: int) : int
  requires { c.val ≥ 0 }
= while (i ≥ 0) do
    invariant { c.val ≥ 0 }
    c.val := c.val+i;
    i := i-1
  done;
  result := c.val
```

```
let m (s:state) (c:reference) (i:int)
  : int diverges
  requires { s.alloct[c] = Cell }
  requires { s.heap.val[c] ≥ 0 }
= let result = ref 0 in
  let c = ref c in
  let i = ref i in
  while (!i ≥ 0) do
    invariant { s.heap.val[!c] ≥ 0 }
    (* c.val := c.val + i *)
    s.heap.val ← Map.add !c
      (s.heap.val[!c]+!i) s.heap.val;
    i := !i-1
  done;
  result := s.heap.val[!c]; !result
```

Fig. 9: Program translation example: WhyRel program on the left, WhyML translation on the right; frame conditions omitted.

written. Hence, WhyRel generates an additional postcondition for method `m`:
`wr_framed_val (old s) s (Rgn.singleton c)`, where

```
predicate wr_framed_val (s: state) (t: state) (r: rgn) = ∀ p: reference.
  s.alloct[p] = Cell ∧ p ∉ r ⇒ s.heap.val[p] = t.heap.val[p]
```

With this postcondition, callers of `m` (in WhyML) can rely on knowing that the `val` fields of only references in {c} are modified.

Biprograms. WhyRel translates biprograms into product programs; specifically, WhyML functions that act on a pair of states[5]. Before translation, it performs an adequacy check to ensure the biprogram is well-formed. Recall that adequacy here means that all computations of the underlying unary programs are covered by their aligned biprogram. Adequacy ensures that a relational judgment about the biprogram entails the expected relation between the underlying unary programs. The check WhyRel performs is syntactic and defined using projection operations on biprograms. Given a biprogram CC, the left projection $\overleftarrow{CC}$ (and resp. the right projection $\overrightarrow{CC}$) extracts the unary program on the left (and resp. the right). As an example, the left projection of $\lfloor$c.f:=g$\rfloor$; (x:=c.f | skip) is c.f:=g; x:=c.f and its right projection is c.f:=g. For adequacy, given unary programs C and C' and their aligned biprogram CC, it suffices to check whether $\overleftarrow{CC} \equiv C$ and $\overrightarrow{CC} \equiv C'$ [1].

Translation of biprograms is described in Fig. 10. The translation function $\mathcal{B}$ takes a biprogram and a pair of contexts (Γ_l, Γ_r) to a WhyML program. In addition to mapping WhyRel identifiers to WhyML identifiers, contexts store information about the state parameters on which the generated WhyML program

---

[5] In reality, generated WhyML functions act on a pair of states and a bijective renaming of references allocated in these states. This is to cater for relation formulas such as $x \doteq y$ where x and y are references. However, this additional parameter is not important to our discussion here, so we avoid mentioning it.

$$\mathcal{B}[\![\texttt{C|C'}]\!](\Gamma_l, \Gamma_r) \qquad\qquad \hateq \mathcal{U}[\![\texttt{C}]\!](\Gamma_l);\ \mathcal{U}[\![\texttt{C'}]\!](\Gamma_r)$$

$$\mathcal{B}[\![\lfloor\texttt{m(x|y)}\rfloor]\!](\Gamma_l, \Gamma_r) \qquad\ \hateq\ apply(\Phi(\texttt{m}), [\Gamma_l.\texttt{st};\Gamma_r.\texttt{st}; \mathcal{E}[\![\texttt{x}]\!](\Gamma_l); \mathcal{E}[\![\texttt{y}]\!](\Gamma_r)])$$

$$\mathcal{B}[\![\lfloor\texttt{C}\rfloor]\!](\Gamma_l, \Gamma_r) \qquad\qquad \hateq \mathcal{B}[\![\texttt{C|C}]\!](\Gamma_l, \Gamma_r)$$

$$\mathcal{B}[\![\texttt{C; C'}]\!](\Gamma_l, \Gamma_r) \qquad\qquad \hateq \mathcal{B}[\![\texttt{C}]\!](\Gamma_l, \Gamma_r);\ \mathcal{B}[\![\texttt{C'}]\!](\Gamma_l, \Gamma_r)$$

$$\mathcal{B}[\![\texttt{var x:T|x:T' in CC}]\!](\Gamma_l, \Gamma_r) \quad \hateq\ \texttt{let } x_l = \texttt{def(T) in let } x_r = \texttt{def(T') in}$$
$$\qquad\qquad \mathcal{B}[\![\texttt{CC}]\!]([\Gamma_l \mid \texttt{x} : x_l], [\Gamma_r \mid \texttt{x} : x_r])$$

$$\mathcal{B}[\![\texttt{if E|E' then CC else DD}]\!](\Gamma_l, \Gamma_r) \hateq\ \texttt{assert } \{\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) = \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r)\};$$
$$\qquad\qquad \texttt{if } \mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \texttt{ then } \mathcal{B}[\![\texttt{CC}]\!] \texttt{ else } \mathcal{B}[\![\texttt{DD}]\!]$$

$$\mathcal{B}[\![\texttt{while E|E' do DD}]\!](\Gamma_l, \Gamma_r) \qquad \hateq\ \texttt{while } \mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \texttt{ do}$$
$$\qquad\qquad \texttt{invariant } \{\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) = \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r)\}$$
$$\qquad\qquad \mathcal{B}[\![\texttt{CC}]\!](\Gamma_l, \Gamma_r)$$

$$\mathcal{B}[\![\texttt{while E|E'. P|P do DD}]\!](\Gamma_l, \Gamma_r)\ \ \hateq$$
$$\qquad \texttt{while } (\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \vee \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r)) \texttt{ do invariant } \{\mathcal{A}\}$$
$$\qquad\quad \texttt{if } (\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \wedge \mathcal{F}[\![\mathcal{P}]\!](\Gamma_l, \Gamma_r)) \texttt{ then } \mathcal{U}[\![\overleftarrow{\texttt{CC}}]\!](\Gamma_l)$$
$$\qquad\quad \texttt{else if } (\mathcal{E}[\![\texttt{E'}]\!](\Gamma_r) \wedge \mathcal{F}[\![\mathcal{P'}]\!](\Gamma_l, \Gamma_r)) \texttt{ then } \mathcal{U}[\![\overrightarrow{\texttt{CC}}]\!](\Gamma_r) \texttt{ else } \mathcal{B}[\![\texttt{CC}]\!](\Gamma_l, \Gamma_r)$$
$$\quad \text{where } \mathcal{A} \equiv (\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \wedge \mathcal{F}[\![\mathcal{P}]\!](\Gamma_l, \Gamma_r))\ \vee (\mathcal{E}[\![E']\!](\Gamma_r) \wedge \mathcal{F}[\![\mathcal{P'}]\!](\Gamma_l, \Gamma_r)) \vee$$
$$\qquad\qquad (\neg\mathcal{E}[\![E]\!](\Gamma_l) \wedge \neg\mathcal{E}[\![E']\!](\Gamma_r)) \vee (\mathcal{E}[\![E]\!](\Gamma_l) \wedge \mathcal{E}[\![E']\!](\Gamma_r))$$
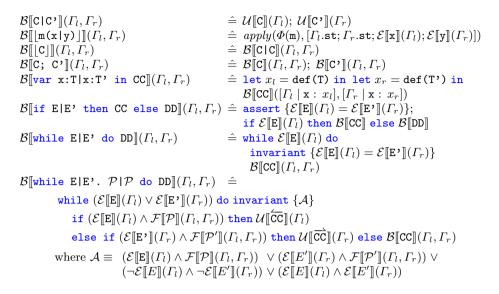
<div align="center">Fig. 10: Translation of biprograms, excerpts</div>

acts. Similar to $\mathcal{B}$, the function $\mathcal{U}$ translates unary programs to WhyML programs, $\mathcal{E}$, expressions to WhyML expressions, and $\mathcal{F}$, a restricted set of relation formulas to WhyML expressions. Biprograms don't require the underlying unary programs to act on a disjoint set of variables; however, this means that WhyRel has to perform appropriate renaming during translation. Renaming is manifest in the translation of variable blocks (`var x:T|x:T' in CC`), where the context Γ_l (and resp. Γ_r) is extended, $[\Gamma_l \mid \texttt{x} : x_l]$, mapping $\texttt{x}$ to a renamed copy x_l (and resp. Γ_r is extended with the binding $\texttt{x} : x_r$).

In translating $(C|C')$, the unary translations of C and C' are sequentially composed. Syncs $\lfloor C \rfloor$ are handled similarly, as syntactic sugar for $(C|C)$, except for the case of method calls. Procedure-modular reasoning about relational properties is enabled by aligning method calls which indicates that the relational spec associated with the method is to be exploited. WhyRel will translate these to calls to the appropriate WhyML product program, using a global method context (Φ in Fig. 10). Since translated product programs act on pairs of states, the generated WhyML call takes $\Gamma_l.\texttt{st}$ and $\Gamma_r.\texttt{st}$, names for left and right state parameters, as additional arguments.

Product constructions for control flow statements require generating additional proof obligations. For aligned conditionals, WhyRel introduces an assertion that the guards are in agreement. Lockstep aligned loops are dealt with similarly; guard agreement must be invariant. For conditionally aligned loops, the generated loop body captures the pattern indicated by the alignment guards $\mathcal{P}|\mathcal{P'}$: if the left (resp. right) guard is true and $\mathcal{P}$ (resp. $\mathcal{P'}$) holds, perform a left-only (resp. right-only) iteration; otherwise, perform a lockstep iteration. Adequacy is ensured

by requiring the condition $\mathcal{A}$ to be invariant. This condition states that until both sides terminate, the loop can perform a lockstep or a one-sided iteration. In relational region logic, the alignment guards $\mathcal{P}$ and $\mathcal{P}'$ can be any relational formula. However, the encoding of conditionally aligned loops is in terms of a conditional that branches on these alignment guards. In Why3, this only works if $\mathcal{P}$ and $\mathcal{P}'$ are restricted; for example, to not contain quantifiers. WhyRel supports alignment guards that include agreement formulas, one-sided points-to assertions, one-sided boolean expressions, and the usual boolean connectives.

Proof obligations for encapsulation. To ensure sound encapsulation, WhyRel performs an analysis on source programs. This analysis includes two parts: a static check to ensure client programs don't directly write to variables in a module's boundary; and the generation of intermediate assertions that express disjointness between the footprints of client heap updates and regions demarcated by module boundaries. For modules with public/private invariants, WhyRel additionally generates a lemma which states that the module's boundary frames the invariant, i.e., the invariant only depends on locations expressed by the boundary. The same is done with coupling relations, for which we need to consider boundaries of both modules being related. A technical condition of relational region logic requiring boundaries grow monotonically as computation proceeds is also ensured by introducing appropriate postconditions in generated programs.

5 Evaluation

We evaluate WhyRel via a series a case studies, representative of the challenge problems highlighted at the outset of this article. Examples include representation independence, optimizations such as loop tiling [5], and others from recent literature on relational verification (including [9] and [21]). Some, like those described in Sec. 3, deal with reasoning in terms of varying alignments including data-dependent ones. Our representation independence examples include showing equivalence of Dijkstra's single-source shortest-paths algorithm linked against two implementations of priority queues, which requires reasoning about fine-grained couplings between pointer structures; and Kruskal's minimum spanning tree algorithm linked against different modules implementing union-find, which requires couplings equating the partitions represented by the two versions. For all examples, VCs are discharged using the SMT solvers Alt-Ergo, CVC4, and Z3. Replaying proofs of most developments using Why3's saved sessions feature takes less than 30 minutes on a machine with an Intel Core i5-6500 processor and 32 gigabytes of RAM.

A primary goal of this work is to investigate whether verifying relational properties of heap manipulating programs can be performed in a manner tractable to SMT-based automation, and for the most part, we believe WhyRel provides a promising answer. The tool serves as an implementation of relational region logic and demonstrates that even its additional proof obligations for encapsulation can be encoded using first-order assertions. In fact, exploration of case studies using WhyRel was instrumental in designing proof rules of relational region logic.

Reasoning about heap effects à la region logic is generally simple and VCs get discharged quickly using SMT. However, technical lemmas WhyRel generates which pertain to showing that module boundaries frame private invariants and couplings require considerable manual effort to prove. These lemmas usually involve reasoning about image expressions, which involve existentials and nontrivial set operations on regions. Given our encoding of states and regions, SMT solvers seem to have difficulties solving these goals. Manual effort involves applying a series of Why3 transformations (or proof tactics) and introducing intermediate assertions. We conjecture that the issue can be mitigated by using specialized solvers [23] or different heap encodings [24].

Another issue with our encoding of typed program states is the generation of a large number of VCs related to well-formedness of states. These account for a substantial fraction of proof replay time. Why3 programs act directly on our minimally-typed state representation and each heap update needs to preserve an invariant that specifies constraints on the types of allocated references (see Fig. 8). Using Why3's support for module abstraction [12] may ameliorate this issue. An alternative is to use assumptions, which can be justified by correctness of the WhyRel type checker and translator.[6]

Apart from these challenges related to verification, we note that specs in region logic tend to be verbose when compared to other formalisms such as separation logic [4].

6   Related work

WhyRel is closely modeled on relational region logic, developed in [1]. That paper provides a high-level overview of WhyRel, using a small set of examples verified in the tool to motivate aspects of the formal logic; but it doesn't give a full presentation of the tool or go into details about the encoding. The paper provides comprehensive soundness proofs of the logic and shows how the VCs WhyRel generates and the checks it performs correspond closely to obligations of relational proof rules. The paper builds on a line of work on region logic [4,2,3]. The VERL tool implements an early version of unary region logic without encapsulation and was used to evaluate a decision procedure for regions [23].

For local reasoning about pointer programs, separation logic is an effective and elegant formalism. For relational verification, ReLoC [13], based on the Iris separation logic and built in the Coq proof assistant supports, apart from many others, language features such as dynamic allocation and concurrency. However, we are unaware of auto-active relational verifiers based on separation logic.

Alignments for relational verification have been explored in various contexts. In WhyRel, the biprogram syntax captures alignment based on control flow, but also caters to data-dependent alignment of loops through the use of alignment guards (as discussed in Sec. 3). Churchill et al. [8] develop a technique for equivalence checking by using data dependent alignments represented by control

---

[6] The Boogie verification language provides "free requires" and "free ensures" syntax for just such assumptions.

flow automata which they use to prove correctness of a benchmark of vectorizing compiler transformations and hand-optimized code. Unno et al. [30] address a wide range of relational problems including k-safety and co-termination, expressing alignments and invariants as constraint satisfaction problems they solve using a CEGIS-like technique. Their work is applied to benchmarks proposed by Shemer et al. [25] who develop a technique for equivalence and regression verification. Both the above works represent alignments as transition systems and perform inference of relational invariants and alignment conditions. Inference relies on solvers and therefore programs need to be restricted so they are amenable to these solvers. A promising approach by Barthe et al. [6] reduces relational verification to proving formulas in trace logic, a multi-sorted first-order logic using first-order provers. In trace logic, conditions can be expressed on traces including relationships between different time points without recourse to alignment per se.

Sousa and Dillig develop Descartes [26] for reasoning about k-safety properties of Java programs automatically using implicit product constructions and in a logic they term Cartesian Hoare logic. Their work is furthered by Pick et al. [22] who develop novel techniques for detecting alignments. The REFINITY [27] workbench based on the interactive KeY tool can be used to reason about transformations of Java programs; heap reasoning relies on dynamic frames and relational verification proceeds by considering *abstract programs*. Other related tools include SymDiff [18] which is based on Boogie and can modularly reason about program differences in a language-agnostic way, and LLRêve [16] for regression verification of C programs. Eilers et al. [10] develop an encoding of product programs for noninterference that facilitates procedure-modular reasoning. They verify a large collection of benchmark examples using the VIPER toolchain.

7   Conclusion

In this paper we present WhyRel, a prototype for relational verification of pointer programs that supports dynamic framing and state-based encapsulation. The tool faithfully implements relational region logic and demonstrates how its proof obligations, including those related to encapsulation, can be encoded in a first-order setting. We've performed a number of representative examples in WhyRel leveraging support Why3 provides for SMT, and believe these demonstrate the amenability of region logic, and its relational variant, to automation.

Data Availability Statement Sources for WhyRel and all examples performed using the tool are available in Zenodo with the identifier https://doi.org/10.5281/zenodo.7308342 [20].

References

1. Banerjee, A., Nagasamudram, R., Naumann, D.A., Nikouei, M.: A relational program logic with data abstraction and dynamic framing. ACM Transactions on Programming Languages and Systems **44**(4), 25:1–25:136 (2022). https://doi.org/10.1145/3551497

2. Banerjee, A., Naumann, D.A.: Local reasoning for global invariants, part II: Dynamic boundaries. Journal of the ACM **60**(3), 19:1–19:73 (2013). https://doi.org/10.1145/2485981

3. Banerjee, A., Naumann, D.A., Nikouei, M.: A logical analysis of framing for specifications with pure method calls. ACM Transactions on Programming Languages and Systems **40**(2), 6:1–6:90 (2018)

4. Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part I: Region logic. Journal of the ACM **60**(3), 18:1–18:56 (2013). https://doi.org/10.1145/2485982

5. Barthe, G., Crespo, J.M., Kunz, C.: Beyond 2-safety: Asymmetric product programs for relational program verification. In: Logical Foundations of Computer Science, International Symposium. Lecture Notes in Computer Science, vol. 7734, pp. 29–43 (2013)

6. Barthe, G., Eilers, R., Georgiou, P., Gleiss, B., Kovács, L., Maffei, M.: Verifying relational properties using trace logic. In: Formal Methods in Computer Aided Design. pp. 170–178 (2019). https://doi.org/10.23919/FMCAD.2019.8894277

7. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. pp. 53–64 (2011)

8. Churchill, B.R., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: ACM Conf. on Program. Lang. Design and Implementation. pp. 1027–1040 (2019)

9. Eilers, M., Müller, P., Hitz, S.: Modular product programs. In: Programming Languages and Systems, European Symposium on Programming. pp. 502–529 (2018)

10. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. **42**(1), 3:1–3:37 (2020). https://doi.org/10.1145/3324783

11. Filliâtre, J.C.: One Logic To Use Them All. In: CADE. Springer (2013)

12. Filliâtre, J.C., Paskevich, A.: Abstraction and genericity in Why3. In: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles. pp. 122–142. Springer International Publishing (2020)

13. Frumin, D., Krebbers, R., Birkedal, L.: ReLoC: A mechanised relational logic for fine-grained concurrency. In: IEEE Symp. on Logic in Computer Science. pp. 442–451 (2018)

14. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Informatica **1**, 271–281 (1972). https://doi.org/10.1007/BF00289507

15. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Formal Methods. Lecture Notes in Computer Science, vol. 4085, pp. 268–283 (2006). https://doi.org/10.1007/11813040_19

16. Kiefer, M., Klebanov, V., Ulbrich, M.: Relational program reasoning using compiler IR: Combining static verification and dynamic analysis. J. Automated Reasoning **60**, 337–363 (2018)
17. Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: ACM CCS (2013)
18. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: Computer Aided Verification. pp. 712–717 (2012). https://doi.org/10.1007/978-3-642-31424-7_54
19. Leino, K.R.M., Moskal, M.: Usable auto-active verification. In: Ball, T., Shankar, N., Zuck, L. (eds.) Usable Verification Workshop (2010), http://fm.csl.sri.com/UV10/submissions/uv2010_submission_20.pdf
20. Nagasamudram, R., Banerjee, A., Naumann, D.A.: The WhyRel Prototype for Modular Relational Verification of Pointer Programs (Nov 2022). https://doi.org/10.5281/zenodo.7308342
21. Naumann, D.A.: Thirty-seven years of relational Hoare logic: Remarks on its principles and history. In: 9th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), Part II. Lecture Notes in Computer Science, vol. 12477, pp. 93–116 (2020), https://doi.org/10.1007/978-3-030-61470-6_7, extended version at https://arxiv.org/abs/2007.06421.
22. Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Computer Aided Verification. pp. 164–182 (2018)
23. Rosenberg, S., Banerjee, A., Naumann, D.A.: Decision procedures for region logic. In: Int'l Conf. on Verification, Model Checking, and Abstract Interpretation. Lecture Notes in Computer Science, vol. 7148, pp. 379–395 (2012)
24. Schmid, G.S., Kuncak, V.: Proving and disproving programs with shared mutable data. CoRR **abs/2103.07699** (2021), https://arxiv.org/abs/2103.07699
25. Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Computer Aided Verification. pp. 161–179 (2019)
26. Sousa, M., Dillig, I.: Cartesian Hoare Logic for verifying k-safety properties. In: ACM Conf. on Program. Lang. Design and Implementation. pp. 57–69 (2016)
27. Steinhöfel, D.: REFINITY to model and prove program transformation rules. In: Asian Symposium on Programming Languages and Systems APLAS. Lecture Notes in Computer Science, vol. 12470, pp. 311–319 (2020). https://doi.org/10.1007/978-3-030-64437-6_16
28. Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments (VSTTE), pp. 496–501 (2008)
29. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 3672, pp. 352–367 (2005)
30. Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 12759, pp. 742–766. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_35

Author Queries

Query Refs.	Details Required	Author's response
AQ1	This is to inform you that corresponding author has been identified as per the information available in the Copyright form.	