WhyRel: An Auto-active Relational Verifier

Ramana Nagasamudram[1], Anindya Banerjee[2], David. A. Naumann[1]

[1]Stevens Institute of Technology, Hoboken, USA.
[2]IMDEA Software Institute, Madrid, Spain.

Contributing authors: rnagasam@stevens.edu; anindya.banerjee@imdea.org; dnaumann@stevens.edu;

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.

1 Introduction

Relational properties encompass conditional equivalence of programs (as in regression verification [1]), noninterference (in which a program is related to itself via a low-indistinguishability relation), and other requirements such as sensitivity [2]. 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. [3–5]).

We aim for auto-active verification [6], 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 have little or no support dynamically allocated objects. So inference is beyond the scope of this article.

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 including encapsulation.

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. WhyRel's 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 [7], a relational extension of region logic [8, 9]. Region logic provides a flexible approach to local reasoning through the use of *dynamic frame* conditions [10] 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 [7], of which our prototype is a faithful implementation.

WhyRel is built on top of the Why3 platform[1] for deductive program verification which provides infrastructure for verifying programs written in WhyML, a subset of ML [11] with support for ghost code and nondeterministic choice. The assertion language is a first-order logic extended with algebraic data types, polymorphism, recursive definitions, and inductive predicates [12]. 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.

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 syntax for *product programs* [13]. 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 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.

This article extends a previous article on WhyRel [14]. Here, we describe new examples, present an improved encoding, and provide additional implementation details.

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 *region* (written `rgn` in code), where a region is a set of references and is used to describe objects notionally owned by modules implementing the stack interface. Variable `capacity` has type `int`

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

```
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(old(self.abs))}
    ensures  {self.abs = tl(old(self.abs))}
    effects  {rw {self}‘any, self.rep‘any, alloc; rd self}
```

Fig. 1: WhyRel interface for the Stack ADT.

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 *location* is either a variable or a heap location $o.f$, where o is an object reference and f is the name of one of its fields. 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 [15].

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` $\in$ `pool`) and stack operations require `self` to be part of the boundary (through pre `self` $\in$ `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. 2a. 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

```
meth prog (n: int) : int
  requires {0 ≤ n < capacity ∧ ...}
  effects  {rw alloc, pool, pool‘any, pool‘rep‘any; rd n, capacity}
= /* Note: locals are initialized to zero-equivalent values */
  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;
```

(a) Example client `prog` of the `STACK` interface.

```
meth prog (n: int|n: int) : (int|int)
  requires { n ≐ n ∧ Both(0 ≤ n < capacity ∧ ... ) }
  ensures  { result ≐ result }
```

(b) Relational specification for `prog` expressing equivalence: the version of `prog` on the left is linked against `ArrayStack`; the version on the right against `ListStack`.

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

left program be the client linked against `ArrayStack`, and the *right* the client linked against `ListStack`. Equivalence is expressed using the relational spec shown in Fig. 2b. For brevity, we omit frame conditions when describing relational specs.

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 ≐ 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[2]. The relational spec requires the two initial states to satisfy the unary precondition for the client, as indicated by `Both(...)`. The post-relation, `result ≐ result`, asserts equality on returned values. In WhyRel, relational specs capture a 2-safety termination-insensitive property: *for any pair of terminating executions of the programs being related, if the initial states are related by the pre-relation then the final states are 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

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

```
meth prog (n: int | n: int) : (int | int)
= var i: int | i: int in var c: Cell | c: Cell in
  var stk: Stack | 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.

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 ⌊`stk := new Stack`⌋. 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 modular verification of relational properties by indicating that relational specs for `push` and `pop`, described later, are to be used.

To prove the spec (in Fig. 2b) 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` $\ddot{=}$ `i` $\wedge$ `stk.abs` $\ddot{=}$ `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` $\ddot{=}$ `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

```
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) }
    requires {Both (self.size > 0) ∧ ...}
    ensures  {... ∧ result.val ≐ result.val} = /* biprogram for pop */
```

Fig. 4: Bimodule for Stack; excerpts.

same; and for pop, which state that given related inputs, the values of the returned `Cell`s 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` $\doteq$ `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` $\doteq$ `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., $\pi(\texttt{x}) = \texttt{y}$.

The relational spec for the initializer `Stack` ensures `self` $\doteq$ `self`, which is required in the specs for `push` and `pop`. Like other invariants, coupling relations 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 on which clients can rely, 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.

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

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

Fig. 5: Two versions of a simple multiplication routine.

```
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.

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. [16] 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` $\doteq$ `n` $\wedge$ `m` $\doteq$ `m` and post-relation `result` $\doteq$ `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 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

```
meth ex (x: int, n: int)
  : int =
  y := x; z := 24; /* 4! */
  w := 0;
  while (y > 4) do
    if (w mod n = 0) then
      z := z * y;
      y := y - 1; end;
    w := w + 1 done;
  result := z;
```

```
meth ex (x: int, n: int)
  : int =
  y := x; z := 16; /* 2^4 */
  w := 0;
  while (y > 4) do
    if (w mod n = 0) then
      z := z * 2;
      y := y - 1; end;
    w := w + 1 done;
  result := z;
```

```
meth ex (x: int, n: int | x: int, n: int) : (int | int) =
  ⌊ y := x ⌋; ( z := 24 | z := 16 ); ⌊ w := 0 ⌋;
  while (y > 4) | (y > 4) . ⟪ w mod n ≠ 0 ⟪ | ⟫ w mod n ≠ 0 ⟫ do
    invariant { ⟪ z ⟪ > ⟫ z ⟫ ∧ y ≐ y ∧ ⟫ y ≥ 4 ⟫ }
    if (w mod n = 0) | (w mod n = 0) then
      ( z := z * y | z := z * 2 );
      ⌊ y := y - 1 ⌋;
    end;
    ⌊ w := w + 1 ⌋;
    done;
  ⌊ result := z ⌋;
```

Fig. 7: Programs that compute the factorial and exponent of $x \geq 4$ and a biprogram.

equality on `result` after the inner loop. In Fig. 6 this is indicated by the assertion after the left inner loop; `old(result)` stands for the value `result` held in the previous iteration (or the value it held prior to entering the loop in case of the first iteration). The notation $\langle\!|P|\!\rangle$ (resp. $|\!\rangle P \langle\!|$) is used to state that the unary formula P holds in the left (and resp. right) state.

Conditionally aligned loops

The previous example exploits the fact that the two loops being related perform the same number of iterations, as implied by their guards being in agreement: i.e., $\langle\!|\texttt{i < n}|\!\rangle = |\!\rangle\texttt{i < n}\langle\!|$. This enables a lockstep alignment of loop iterations, which in turn leads to simple relational invariants. In many situations, however, lockstep reasoning is inapplicable. For these, WhyRel provides for other patterns of loop alignment, including those that account for conditions on data.

Consider the fairly contrived example shown in Fig. 7 (adapted from [17]). The version of `ex` on the left computes the factorial of `x` for $\texttt{x} \geq 4$ and the version on the right computes the exponent `2^x`. We aim to prove that for $\texttt{x} \geq 4$, factorial majorizes exponent, as expressed by the following relational spec. The post-relation in the `ensures` clause says the value of `result` on the left is strictly greater than the value of `result` on the right.

```
meth ex (x: int, n: int | x: int, n: int) : (int | int)
  requires { x ≐ x ∧ ⟫ x ≥ 4 ⟫ ∧ Both (n > 0) }
  ensures  { ⟪ result ⟪ > ⟫ result ⟫ }
```

The implementations of `ex` are structured similarly. To keep the example simple, the two versions start by setting `z` to be 4! and 2^4 respectively. The variable `w` is used to add *stuttering* steps: both loops update `z` and the counter `y` only when `w` is a multiple of `n`. The only constraint on `n` in the relational spec for `ex` is that it must be strictly greater than 0 on both sides.

A purely lockstep alignment of the two loops doesn't work anymore since one side may perform more iterations than the other. Instead, we want to align iterations in lockstep only when both sides are about to update `z` and `y`, i.e., when `w mod n` is 0 on both sides. In case an iteration on a side is not going to update `z`, we want to align this iteration with doing nothing, or `skip`, on the other side. Lockstep and one-sided iterations of this kind preserve the relation $\langle\!|\texttt{z}|\!\rangle > \langle\!|\texttt{z}|\!\rangle$.

Syntactically, the alignment is captured by the biprogram shown in Fig. 7. Conditional alignment is captured through the use of additional annotations, called *alignment guards*. These are general relation formulas that express conditions that lead to left-only, right-only, or lockstep iterations. The alignment guards of the biprogram for `ex` are $\{\!|\ \texttt{w mod n} \neq \texttt{0}\ |\!\}\ |\ \{\!|\ \texttt{w mod n} \neq \texttt{0}\ |\!\}$. The left alignment guard $\{\!|\ \texttt{w mod n} \neq \texttt{0}\ |\!\}$ says that the biprogram performs left-only iterations in the case that `w` on the left is *not* a multiple of `n` on the left. A left-only iteration has the effect of executing `if (w mod n = 0) then (z := z*y; y := y-1) end; w := w+1` on the left. Since this iteration is guarded by the left alignment guard, the overall effect is to simply increment `w` on the left. The right alignment guard is similar to the left and specifies the condition under which to consider right-only iterations. Iterations are considered in lockstep when both alignment guards are false, i.e., in the case in which `w` is a multiple of `n` on both sides. With this alignment, the relational invariant shown in Fig. 7 suffices to establish the post-relation of interest: $\{\!|\ \texttt{result}\ |\!\} > \{\!|\ \texttt{result}\ |\!\}$.

The version of this example developed in [17] fixes `n` to be 2 on the left and 3 on the right. The alignment required for fixed `n` can be expressed in terms of loop unfoldings, so in principle it may be within reach of some automated techniques [16, 18, 19].[3] However, such forms of alignment are not adequate for our version where `n` is not fixed, nor for the following example.

Another example of conditional alignment

A less contrived example demonstrating conditional alignment of loop iterations is shown in Fig. 8. The `sumpub` program traverses a linked list and computes the sum of all elements marked public, as indicated by each element's `pub` field. The program satisfies a noninterference property: its output doesn't depend on the values of non-public elements of the input list. This property is specified as follows.

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

Here, `listpub(l,xs)` says that the *sequence* of public elements reachable from the list pointer `l` is realized in `xs`, a mathematical list of integers. The pre-relation requires the two runs of `sumpub` being related start with lists with the same sequence of public values. The post-relation ensures that the two runs produce the same result.

[3] However, [19] specifically mentions this example as a benchmark that their tool currently cannot verify.

```
class Node {
  pub: bool;
  data: int;
  nxt: Node
}
class List {
  head: Node;
  rep: rgn
}
```

```
meth sumpub (l: List) : int =
  p := l.head; sum := 0;
  while (p ≠ null) do
    if p.pub then
      sum := sum + p.data
    end;
    p := p.nxt
  done;
  result := sum;
```

```
/* Biprogram for two copies of sumpub */
meth sumpub (l: List | l: List) : int =
  ⌊ p:=l.head ⌋; ⌊ sum:=0 ⌋;
  while (p ≠ null) | (p ≠ null) .
    ⦇ ¬ p.pub ⦈ | ⦇ ¬ p.pub ⦈ do
    ( if p.pub then sum:=sum+p.data end;
      p:=p.nxt
    | if p.pub then sum:=sum+p.data end;
      p:=p.nxt)
  done; ⌊ result:=sum ⌋;
```

Fig. 8: Summing up public elements of a linked list.

```
inductive listpubN (n: Node, xs: intList) =
  | lp_nil : listpubN null nil
  | lp_nxt : ∀ n xs.
    ¬ n.pub ⇒ listpubN n.nxt xs ⇒ listpubN n xs
  | lp_add : ∀ n xs.
    n.pub ⇒ listpubN n.nxt xs ⇒ listpubN n (n.data::xs)

predicate listpub (l: List, xs: intList) =
  l ≠ null ⇒ listpubN(l.head, xs)
```

We can prove that the program computes exactly the sum of public elements: `result = sum(xs)`; this would imply the desired noninterference property. We can also come up with a simpler proof by conditionally aligning the loops in the two copies of `sumpub` being related. The alignment we use is given by the biprogram in Fig. 8. It works as follows: if `p` is a non-public node on one side, perform a loop iteration on the other side; and if `p` on both sides is public, align the two loop bodies in lockstep. This strategy has the effect of incrementing `sum` exactly when both sides are visiting public nodes, the values of which are guaranteed to be the same by the relational precondition.

This biprogram maintains $\exists$ `xs|xs.` `Both(listpub(p,xs))` $\wedge$ `xs`$\doteq$`xs` $\wedge$ `s`$\doteq$`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 `sum` 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 `sum` 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

WhyRel functions as a front-end to Why3, translating source programs to WhyML which serves as an *intermediate verification language*. This design is similar to other tools such as Dafny based on Boogie [20], Frama-C based on Why3 [21], and Nagini based on Viper [22]. During translation, WhyRel performs a number of checks and transformations: primary among these are checks that clients of modules respect encapsulation and that user provided biprograms are adequate. Proof obligations pertinent to relational region logic, including encapsulation, are encoded as intermediate assertions in WhyML programs and as lemmas for the user to prove. WhyRel is implemented in OCaml and relies on a library provided by Why3 for constructing and pretty-printing WhyML parse trees.

Encoding program states

WhyML supports records with mutable fields, encompassing ML-style references. A static analysis is used to disallow any aliasing [11, 23]. It helps ensure that the verification conditions for WhyML are simple enough to be amenable to automation. It also means that Why3 does not have to fix a memory model or an assertion language with explicit support for the heap, for e.g., one based on a separation logic. To overcome Why3's limitations on aliasing and faithfully represent the heap model formalized in region logic, WhyRel generates WhyML programs that act on an explicit representation of program state.

Our encoding models WhyRel references as an uninterpreted WhyML type `reference` with a distinguished element, `null`. WhyRel permits equality comparisons on reference values, but not pointer-arithmetic. Regions in WhyRel are encoded using WhyML ghost state, as mathematical sets of `reference`s. The usual set operations are supported. In addition, WhyRel axiomatizes image expressions: for each field f, WhyRel generates a WhyML function symbol `img_f` along with an axiom that captures the meaning of $G`f$ for any given region G.

WhyRel program states are encoded using WhyML records and a usual fields-as-arrays representation [24]. An example is shown in Fig. 9. The `state` type includes one component for each field of a class in the source program. The component `alloct` stores a map from references to object types, `reftype`, and is used to keep track of allocated objects. Each source language field is modeled as a map from references to values. The set of values includes references, WhyML mathematical types such as arrays and lists, regions, and primitives 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 type `Rgn.t` is the type of regions in WhyML. Our encoding of program states operates on whole programs, so WhyRel requires access to all class definitions in all modules. At the WhyML level, WhyRel classes and fields are treated globally.

The `state` type is annotated with a WhyML invariant that captures well-formedness of program states. In addition to conditions such as `null` never being allocated, and there being no dangling references, this invariant captures constraints related to typing: for example, the value stored in the `nxt` field of a `Node` is either `null` or allocated at type `Node`. For the example shown in Fig. 9, the invariant is of the form:

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

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

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

```
type reftype = Cell | Node (*class names*)
type state = {
  (* fields *)
  mutable data: map reference int;
  mutable ghost rep: map reference Rgn.t;
  mutable curr: map reference reference;
  mutable nxt: map reference reference;
  (* global variables *)
  mutable alloct: map reference reftype;
  mutable ghost pool: Rgn.t }

(* axiomatization of r'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.nxt[q]
```

Fig. 9: State encoding: WhyRel source on left, encoding in WhyML on right.

```
type state = { ... }
  (* null is never allocated *)
  invariant { ¬(mem null alloct) }
  (* null does ¬ have any fields *)
  invariant { ¬(mem null data) ∧ ... }
  (* type conditions for objects of type Node *)
  invariant { ∀ p. alloct[p] = Node → mem p curr ∧ mem p nxt
    ∧ (curr[p] = null ∨ alloct[curr[p]] = Cell)
    ∧ (nxt[p]  = null ∨ alloct[nxt[p]]  = Node) }
  invariant { ... }
```

Finally, for each field of the source language, WhyRel generates a setter in WhyML, used in modeling field updates. The setter for field `nxt` is shown below.

```
val set_nxt (s: state) (p: reference) (q: reference) : unit
  requires {s.alloct[p] = Node ∧ (q = null ∨ s.alloct[q] = Node)}
  ensures {s.nxt = add (old s.nxt) p q}
  writes {s.nxt}
```

Definitions relevant to our state encoding are placed in a WhyML module named `State`, imported by all programs generated by WhyRel. The `State` module additionally includes, for each class `K` in the source program, a function `mk_K` that allocates a new object of type `K` in a given state.

Translating unary programs and effects

WhyRel translates unary programs to WhyML functions that act on our encoding of states. Commands that modify the heap, i.e., field updates, are modeled as calls to the appropriate setters. Local variables, parameters, and the distinguished `result` variable are encoded using WhyML `ref` cells.[4] Object parameters are modeled using the `reference` type along with a typing assumption. Translation of control flow statements

[4]That is, using ML-style references.

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

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

Fig. 10: Program translation example: WhyRel program on the left, WhyML translation on the right; frame conditions omitted.

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. 10 shows an example translation. Note that (`:=`) is used to update `ref` cells and (`!`) to dereference `ref` cells.

Translation of frame conditions requires care given our encoding of states. As an example, the writes for method `m` shown in Fig. 10 would include `rw {c}‘data` due to the write to, and read of, field `data` of object `c`. Correspondingly, in the Why3 translation of `m`, component `data` of the state parameter `s` is updated via the setter `set_data`; so specifying the function in Why3 requires adding `writes {s.data}` as annotation. However, this doesn't describe framing at the granularity we desire: it implies that the field `data` of any reference whatsoever can be written. To deal with this, WhyRel generates a specification for method `m` with an additional postcondition: `wframed_data (old s) s (Rgn.singleton c)`, where

```
predicate wframed_data (s: state) (t: state) (r: rgn) =
  ∀ p: reference.
    s.alloct[p] = Cell ∧ ¬ (Rgn.mem p r) ⇒ s.data[p] = t.data[p]
```

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

Biprograms

WhyRel translates biprograms into product programs: WhyML functions that act on a pair of states.[5] Before translation, WhyRel performs an adequacy check on biprograms. Intuitively, a biprogram is adequate if its two underlying programs are related in accord with a relational spec whenever the biprogram satisfies that spec. That is, correctness of the biprogram implies relatedness.

[5]WhyML functions that encode biprograms also act on a refperm: a renaming of references allocated in the two states being related. Refperms are used when translating relation formulas such as $x \doteq y$ where x and y are references [7]. To streamline discussion, we avoid mention of refperms here.

$$
\begin{aligned}
&\mathcal{B}[\![\texttt{C|C'}]\!](\Gamma_l,\Gamma_r) && \triangleq\ \mathcal{U}[\![\texttt{C}]\!](\Gamma_l);\ \mathcal{U}[\![\texttt{C'}]\!](\Gamma_r)\\
&\mathcal{B}[\![\lfloor\texttt{m(x|y)}\rfloor]\!](\Gamma_l,\Gamma_r) && \triangleq\ \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) && \triangleq\ \mathcal{B}[\![\texttt{C|C}]\!](\Gamma_l,\Gamma_r)\\
&\mathcal{B}[\![\texttt{CC; DD}]\!](\Gamma_l,\Gamma_r) && \triangleq\ \mathcal{B}[\![\texttt{CC}]\!](\Gamma_l,\Gamma_r);\ \mathcal{B}[\![\texttt{DD}]\!](\Gamma_l,\Gamma_r)\\
&\mathcal{B}[\![\texttt{var x:T|x:T' in CC}]\!](\Gamma_l,\Gamma_r) && \triangleq\ \texttt{let}\ x_l = \texttt{def(T)}\ \texttt{in let}\ x_r = \texttt{def(T')}\ \texttt{in}\\
&&&\quad \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) && \triangleq\ \texttt{assert}\ \{\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) = \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r)\};\\
&&&\quad \texttt{if}\ \mathcal{E}[\![\texttt{E}]\!](\Gamma_l)\ \texttt{then}\ \mathcal{B}[\![\texttt{CC}]\!](\Gamma_l,\Gamma_r)\\
&&&\quad \texttt{else}\ \mathcal{B}[\![\texttt{DD}]\!](\Gamma_l,\Gamma_r)\\
&\mathcal{B}[\![\texttt{while E|E' do DD}]\!](\Gamma_l,\Gamma_r) && \triangleq\ \texttt{while}\ \mathcal{E}[\![\texttt{E}]\!](\Gamma_l)\ \texttt{do}\\
&&&\quad \texttt{invariant}\ \{\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) = \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r)\}\\
&&&\quad \mathcal{B}[\![\texttt{CC}]\!](\Gamma_l,\Gamma_r)\\
&\mathcal{B}[\![\texttt{while E|E'. }\mathcal{P}\texttt{|}\mathcal{P}\texttt{ do DD}]\!](\Gamma_l,\Gamma_r) && \triangleq\\
&\quad \texttt{while}\ (\mathcal{E}[\![\texttt{E}]\!](\Gamma_l) \vee \mathcal{E}[\![\texttt{E'}]\!](\Gamma_r))\ \texttt{do invariant}\ \{\mathcal{A}\}\\
&\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)\\
&\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\\
&&&\quad (\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))
\end{aligned}
$$

Fig. 11: Translation of biprograms, excerpts.

Adequacy holds when a biprogram can cover all pairs of executions of the underlying programs. WhyRel checks this in two stages. The first stage is syntactic and uses projection operations to ensure that the biprogram is indeed constructed from the unary programs of interest. For a biprogram CC, its left projection $\overleftarrow{CC}$ (and resp. its right projection $\overrightarrow{CC}$) is the unary program on the left (and resp. on 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`. Given unary programs C and C' and their aligned biprogram CC, WhyRel checks that $\overleftarrow{CC} \equiv C$ and $\overrightarrow{CC} \equiv C'$, where $(\equiv)$ is syntactic equality. In the second stage, WhyRel adds annotations—assertions or loop invariants—to the biprogram which serve to ensure adequacy. For example, a biprogram that aligns two loops in lockstep includes as relational loop invariant the fact that the two loop guards are in agreement.

Translation of biprograms is described in Fig. 11. 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 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

```
let sumpub l_s r_s (l_l: reference) (r_l: reference) : (int, int)
  requires { l_s.alloct[l_l] = List ∧ r_s.alloct[r_l] = List }
  requires { ∃ l_xs: intList, r_xs: intList.
    listpub l_s l_l l_xs ∧ listpub r_s r_l r_xs ∧ l_xs = r_xs }
  ensures  { match result with (l_res, r_res) → l_res = r_res end }
= ... (* set up locals; translation of stmts before loop *)
  while (!l_p ≠ null || !r_p ≠ null) do
    invariant { ... (* those provided in the biprogram *) }
    invariant {
      (* adequacy condition *)
         (!l_p ≠ null ∧ l_s.pub[!l_p] = false)
       ∨ (!r_p ≠ null ∧ r_s.pub[!r_p] = false)
       ∨ (!l_p ≠ null ∧ !r_p ≠ null) ∨ (!l_p = null ∧ !r_p = null) }

    if (!l_p ≠ null ∧ l_s.pub[!l_p] = false) then
      (* left-only iteration when
         left loop guard and left alignment guard hold. *)
      if l_s.pub[!l_p] then l_sum := !l_sum + l_s.data[!l_p];
      l_p := l_s.nxt[!l_p]
    else if (!r_p ≠ null ∧ r_s.pub[!r_p] = false) then
      (* right-only iteration when
         right loop guard and right alignment guard hold. *)
      ...
    else
      (* lockstep iteration otherwise *)
      if l_s.pub[!l_p] then l_sum := !l_sum + l_s.data[!l_p];
      l_p := l_s.nxt[!l_p];
      if r_s.pub[!r_p] then r_sum := !r_sum + r_s.data[!r_p];
      r_p := r_s.nxt[!r_p];
  done; ...
```

Fig. 12: Example biprogram translation.

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, $[\Gamma_l \mid \mathtt{x} : x_l]$, mapping `x` to a renamed copy x_l (and resp. Γ_r is extended with the binding $\mathtt{x} : x_r$).

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. 11). Since translated product programs act on pairs of states, the generated WhyML call takes Γ_l.`st` and Γ_r.`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 $\mathcal{P} \mid \mathcal{P}'$: 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.

The WhyML program corresponding to the `sumpub` biprogram in Fig. 8 is shown in Fig. 12. The function parameters `l_s` and `r_s` are the left and the right state parameters; `l_l` and `r_l` model the parameter `l` of type `List` in Fig. 8 on either side. The body of the WhyML product program is a loop which encodes the biprogram loop shown in Fig. 8. This loop consists of a nested conditionals which branch on the left and right loop and alignment guards. For example, in the case where the left loop guard is true and the left alignment guard $\langle\!|\neg$ `p.pub`$|\!\rangle$ holds, the WhyML product performs a left-only iteration. The adequacy condition is added as a loop invariant by WhyRel during translation.

Proof obligations for encapsulation

To ensure that clients of modules respect 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. An example is given in Fig. 13.

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 [25], and others from recent literature on relational verification (including [26] and [17]). 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

```
/* WhyRel source */
interface I =
  class C {f: int; rep: rgn;}
  public pool : rgn;
  /* this boundary specifies encapsulated locations */
  boundary { pool, pool‘rep, pool‘rep‘any }
end

module Client =
  import I
  meth main () : unit = /* frame conditions elided */
    var c: C in
    c := new C;
    c.f := 0;
    /* Writes to locations in the boundary of I are disallowed */
    /* pool := {} */ /* raises an error */
end

(* WhyML program corresponding to main *)
let main (s: state) : unit =
  let c = ref null in
  c := mk_C s;  (* allocate a new object of type C in state s *)
  (* assertion added by WhyRel's encap check
     ensures c isn't contained in the boundary of I *)
  assert { ¬ (Rgn.mem c (Rgn.union pool (img_rep s s.pool))) };
  set_f s !c 0; (* translation of c.f := 0 *)
```

Fig. 13: An example of WhyRel's encapsulation check.

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, WhyRel generates some technical lemmas which require considerable manual effort to prove; these lemmas ensure that private invariants and couplings are framed by module boundaries. 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 [27] or different heap encodings [28].

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 [8].

6 Related work

WhyRel is closely modeled on relational region logic, developed in [7]. That article 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. It 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 article builds on a line of work on region logic [8, 9, 29]. The VERL tool implements an early version of unary region logic without encapsulation and was used to evaluate a decision procedure for regions [27]. A previous paper on WhyRel describes the tool and a few select case studies [14]. The current article extends it and includes additional details on case studies and tool implementation. The encoding of program states detailed in Sec. 4 differs from the encoding described in [14] and helps reduce the overall verification burden on users. Previously, WhyRel would translate field updates to direct writes to the `state` parameter. The user would have to prove the `state` invariant is preserved after each of these writes. This is no longer the case thanks to the judicious use of setters. The encoding may be improved further by exploiting Why3's features for abstraction [30].

For local reasoning about pointer programs, separation logic is an effective and elegant formalism. For relational verification, ReLoC [31], 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. [16] develop a technique for equivalence checking by using data dependent alignments represented by control flow automata which they use to prove correctness of a benchmark of vectorizing compiler transformations and hand-optimized code. Unno et al. [32] 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 counterexample-guided inductive synthesis (CEGIS)-like technique. Their work is applied to benchmarks proposed by Shemer et al. [5] 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. Itzhaky et al. [33] more tightly integrate solving for alignments with solving for invariants. Since inference relies on solvers, the techniques are most effective on programs that can be encoded in ways amenable to the solvers. Currently this does not include heap based data structures.

More recently, Dickerson et al. [19] develop a technique for searching for good alignments represented in a form similar to our biprograms. They use e-graphs to compactly represent the space of all possible alignments of two programs, and use sample executions to guide the search. An algebraic view of alignment is also explored by Antonopoulos et al. [34]. These techniques may lead to ways for an auto-active tool to help the user explore alignments.

A promising approach by Barthe et al. [2] 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 [35] 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. [36] who develop novel techniques for detecting alignments. The REFINITY [37] 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 [38] which is based on Boogie and can modularly reason about program differences in a language-agnostic way, and LLRêve [39] for regression verification of C programs. Eilers et al. [40] 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. Eilers et al. [41] explore the advantages of product constructions at the intermediate verification language level, such as in WhyML, over constructions at the source level. Recent works address the verification of $\forall\exists$ relational properties which encompass relations such as refinement and formulations of noninterference and go beyond what WhyRel supports [18, 32, 33].

7 Conclusion

In this article 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.

Acknowledgments. We thank the anonymous TACAS 2023 reviewers and artifact evaluators for their thorough feedback and suggestions which have led to major improvements in this article. We also thank Lennart Beringer for detailed feedback and thoughts on improving the article. We thank Seyed Mohammad Nikouei who built an initial version of WhyRel which helped guide the design of the current version. Nagasamudram and Naumann were partially supported by NSF award 1718713. Banerjee's research was based on work supported by the NSF, while working at the Foundation. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not necessarily reflect the views of the NSF.

Data Availability Statement. The current development version of WhyRel can be found at https://github.com/dnaumann/RelRL. The previous version of the tool (described in [14]), including sources and Why3 session files for all case studies, can be found on Zenodo with identifier https://doi.org/10.5281/zenodo.7308342 [42].

References

[1] Strichman, O., Godlin, B.: Regression verification - a practical way to verify programs. In: Verified Software: Theories, Tools, Experiments (VSTTE), pp. 496–501 (2008)

[2] 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

[3] 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)

[4] Kovács, M., Seidl, H., Finkbeiner, B.: Relational abstract interpretation for the verification of 2-hypersafety properties. In: ACM Conference on Computer and Communications Security, pp. 211–222 (2013). https://doi.org/10.1145/2508859.2516721

[5] Shemer, R., Gurfinkel, A., Shoham, S., Vizel, Y.: Property directed self composition. In: Computer Aided Verification, pp. 161–179 (2019)

[6] 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

[7] 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), 25–125136 (2022) https://doi.org/10.1145/3551497

[8] Banerjee, A., Naumann, D.A., Rosenberg, S.: Local reasoning for global invariants, part I: Region logic. Journal of the ACM **60**(3), 18–11856 (2013) https://doi.org/10.1145/2485982

[9] Banerjee, A., Naumann, D.A.: Local reasoning for global invariants, part II: Dynamic boundaries. Journal of the ACM **60**(3), 19–11973 (2013) https://doi.org/10.1145/2485981

[10] 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

[11] 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)

[12] Filliâtre, J.: One logic to use them all. In: Int'l Conf. on Automated Deduction. Lecture Notes in Computer Science, vol. 7898, pp. 1–20 (2013). https://doi.org/10.1007/978-3-642-38574-2_1

[13] Barthe, G., Crespo, J.M., Kunz, C.: Relational verification using product programs. In: Formal Methods. Lecture Notes in Computer Science, vol. 6664 (2011)

[14] Nagasamudram, R., Banerjee, A., Naumann, D.A.: The WhyRel prototype for modular relational verification of pointer programs. In: Tools and Algorithms for the Construction and Analysis of Systems. Lecture Notes in Computer Science, vol. 13994, pp. 133–151 (2023). https://doi.org/10.1007/978-3-031-30820-8_11

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

[16] 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)

[17] 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). Extended version at https://arxiv.org/abs/2007.06421. https://doi.org/10.1007/978-3-030-61470-6_7

[18] Beutner, R.: Automated software verification of hyperliveness. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 196–216 (2024)

[19] Dickerson, R., Mukherjee, P., Delaware, B.: Kestrel: Relational verification using e-graphs for program alignment. CoRR **abs/2404.08106** (2024) https://doi.org/10.48550/ARXIV.2404.08106 2404.08106

[20] Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming Artificial Intelligence and Reasoning, pp. 348–370 (2010). https://doi.org/10.1007/978-3-642-17511-4_20

[21] Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., Williams, N.: The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. Commun. ACM **64**(8), 56–68 (2021) https://doi.org/10.1145/3470569

[22] Eilers, M., Müller, P.: Nagini: A static verifier for python. In: Computer Aided Verification, pp. 596–603 (2018)

[23] Filliâtre, J.-C., Gondelman, L., Paskevich, A.: A Pragmatic Type System for Deductive Verification. working paper or preprint (2016). https://inria.hal.science/hal-01256434

[24] Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction, pp. 102–126 (2000)

[25] 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)

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

[27] 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)

[28] Schmid, G.S., Kuncak, V.: Proving and disproving programs with shared mutable data. CoRR **abs/2103.07699** (2021) 2103.07699

[29] 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), 6–1690 (2018)

[30] Filliâtre, J.-C., Paskevich, A.: Abstraction and genericity in Why3. In: Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles, pp. 122–142 (2020)

[31] 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)

[32] Unno, H., Terauchi, T., Koskinen, E.: Constraint-based relational verification. In: Computer Aided Verification. Lecture Notes in Computer Science, vol. 12759, pp. 742–766 (2021). https://doi.org/10.1007/978-3-030-81685-8_35

[33] Itzhaky, S., Shoham, S., Vizel, Y.: Hyperproperty verification as CHC satisfiability. In: Programming Languages and Systems, European Symposium on Programming. Lecture Notes in Computer Science, vol. 14577, pp. 212–241 (2024). https://doi.org/10.1007/978-3-031-57267-8_9 . https://doi.org/10.1007/978-3-031-57267-8_9

[34] Antonopoulos, T., Koskinen, E., Le, T.C., Nagasamudram, R., Naumann, D.A., Ngo, M.: An algebra of alignment for relational verification. Proc. ACM Program. Lang. **7**(POPL) (2023) https://doi.org/10.1145/3571213

[35] 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)

[36] Pick, L., Fedyukovich, G., Gupta, A.: Exploiting synchrony and symmetry in relational verification. In: Computer Aided Verification, pp. 164–182 (2018)

[37] 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

[38] 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

[39] 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)

[40] Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. **42**(1), 3–1337 (2020) https://doi.org/10.1145/3324783

[41] Eilers, M., Meier, S., Müller, P.: Product programs in the wild: Retrofitting program verifiers to check information flow security. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification, pp. 718–741. Springer, Cham (2021)

[42] Nagasamudram, R., Banerjee, A., Naumann, D.A.: The WhyRel Prototype for Modular Relational Verification of Pointer Programs. Zenodo (2022). https://doi.org/10.5281/zenodo.7308342