# WhyRel: A Prototype for Relational Verification

Ramana Nagasamudram

Stevens Institute of Technology
rnagasam@stevens.edu

## 1 Motivation

In verifying a behavioral property of a program, it might not always be sufficient to consider a single run or execution. For example, proving noninterference, the independence of a program's public outputs and its private inputs, requires reasoning about two executions that start from public-equivalent states. *Relational properties* describe relations between multiple runs of programs and subsume ordinary program properties [13]. Of particular interest is the relational property of observational equivalence: any two runs of $C$ and $D$, possibly different programs, that start in the same state, end in the same state. Reasoning about observational equivalence can naturally be applied to establish correctness of program transformations, to argue equivalence of library implementations, or to perform regression verification, wherein a program is shown equivalent to a revised version.

Relational Hoare logics (RHLs) provide foundational frameworks for reasoning about relational properties of two programs [7]. Like unary Hoare logics, they include compositional proof rules that help decompose verification; proofs usually relate similary structured subprograms, and verification effort grows as a function of the difference between the two programs being considered. In this research abstract, we describe ongoing work to build a verification tool based on *Relational Region Logic*, a relational program logic for object-based programs that formalizes encapsulation and data abstraction [1, 3]. The expected contribution is a verification system that can be used to evaluate the applicability of this program logic.

## 2 Introduction to WhyRel

WhyRel is a prototype verification tool for an imperative language with objects, dynamic allocation, and shared mutable state. It implements relational region logic, using verification conditions to capture core reasoning principles. This logic combines a deductive system for relational verification with a program logic for verifying partial correctness of heap manipulating programs. The specification language is standard first-order logic, augmented, for relational reasoning, with constructs for relating values of variables in two different program states. Relational region logic also formalizes a semantics for encapsulation, and provides sound proof rules for verification in the presence of information hiding.

On top of this logic, WhyRel adds a light weight module system with interfaces, support for user-defined classes, and commonly used data structures such as arrays. For unary verification, users provide modules with class definitions, method contracts and implementations, along with annotations such as assertions and loop invariants. Modules can contain lemmas and private invariants, which are hidden from clients. For relational verification, users provide relational contracts that tie in two implementations and indicate a desired (relational) proof strategy by providing *biprograms* (explained below). Additionally, WhyRel includes special support for reasoning about equivalence of encapsulated data representations—users can define simulation relations between two modules implementing the same interface, and the tool can be used to generate relational specifications for equivalence of module methods. Given a client that relies on this interface, WhyRel uses a rule of the logic to derive a proof that the behavior of this client is preserved when linked against either module implementation.

Verification is primarily done in an automated fashion, leveraging SMT solvers, though interactive proofs can be performed when needed. We now highlight key aspects of reasoning using WhyRel.

Reasoning about programs that manipulate the heap is tractable when done locally; one should only concern with the actual cells modified, not with changes to unrelated locations due to aliasing [14]. WhyRel inherits support for local reasoning from *unary region logic* [2,4]. The central abstraction is of a *region*: a finite set of references. Programs are associated with *effects*, described as writes to, or reads of, locations in regions. This enables reasoning about preservation. For example, knowing $C$ only writes to locations in a region $g$ lets us conclude that it preserves the value of any $y \notin g$, and the truth of any predicate that only depends on locations in $g$. A large number of reasoning patterns can be captured in this manner; and disjointness of regions, in particular, facilitates proofs based on separation. Importantly, frame based reasoning about the heap can be done in a first-order logic without non-standard operators or extensions.

A well studied approach to the task of relational verification is to construct a *product program* that interleaves, or aligns, executions of the two programs in consideration [5,6]. A combination of unary proof rules and specialized relational proof rules for products can be then be used for verification. In WhyRel, alignment is indicated using *biprograms*. To verify a relational spec about $C$ and $D$, the user starts with a biprogram, denoted $(C|D)$, capturing the following alignment strategy—execute $C$ followed by $D$. This biprogram can then be *woven* into a more aligned version, covenient for verification. For example, loops in $C$ and $D$ can be aligned, allowing for correctness arguments that start by relating loop bodies using (simple) relational loop invariants. Through this process, the task of proving a relational spec can be decomposed naturally and similarities between the two programs can be exploited. In many cases, proofs of functional correctness, which might become complicated, can be avoided.

WhyRel includes support for heap encapsulation, hiding of module invariants, and defining coupling relations between two implementations of a module. Together, these features provide for a systematic approach to verify data abstraction—preservation of client program behavior under change in a module's internal representations [12]. Encapsulation is primarily at the granularity of modules. Each module designates a *dynamic boundary*, a region containing internal locations, and WhyRel checks that client programs respect this boundary. Private invariants, i.e. invariants on encapsulated state, can be hidden from clients provided they hold after module initialization. This liberates client reasoning from reliance on module representations. Coupling relations facilitate proving equivalence of modules that implement the same interface using different data representations. Relational region logic provides sound proof rules that capture these reasoning principles. WhyRel can be used to prove correctness of data representations and equivalence of client programs linked against equivalent modules.

*Architecture.* WhyRel is built on top of the Why3 platform for deductive program verification [8]. Why3 provides infrastructure to verify programs, written in a fragment of ML (WhyML) augmented with ghost code; using a polymorphic first-order logic extended with algebraic data types, recursive definitions, and inductively defined predicates [11]. Verification conditions generated by Why3 can be discharged using a wide array of tools from interactive and automated theorem provers, to SMT solvers. Our tool compiles source programs to WhyML, relying on Why3 to provide facilities for verification and user-interaction.

Since WhyML does not support shared references, biprograms, and hiding of module invariants, significant encoding has to be done by our translation. Program states are deeply embedded, with references as an abstract uninterpreted type; biprograms are translated to product programs that act on a pair of states; and additional proof obligations are generated to ensure encapsulation is respected and information hiding can be done soundly. To help with reasoning in the region logic framework, WhyRel includes a standard library with auxiliary lemmas, imported by all developments.

## 3 Current Progress and Future Plans

WhyRel is a work in progress, and has been used to verify a number of representative examples. The current implementation is open-source and publicly available[1]. Our most challenging case studies so far are exercises in verifying data abstraction. In one, we prove equivalence of two implementations of the Union-Find data structure using a coupling relation that posits a shared abstraction—both implementations represent the same partition of a set. We then formally argue that the behavior of Kruskal's minimum spanning tree algorithm is preserved when linked against either implementation. A similar case study has been completed for Dijkstra's shortest-path algorithm linked against two implementations of Priority Queues. WhyRel has also been used to verify a limited number of examples from recent literature [9, 10, 13].

[1] `https://github.com/dnaumann/RelRL`

Our short term goal is to perform additional case studies in WhyRel, prioritizing those that deal with program transformations. We also plan to investigate static analyses that can obviate the need to generate proof obligations for certain side conditions of the rules in relational region logic. Finally, we hope to add to this program logic a rule for rewriting: to relate $C$ and $D$, at some relational spec, it should be sufficient to rewrite $C$ into an (unconditionally) equivalent program $C'$, and prove the spec for $C'$ and $D$. Such a rule, with an implementation in WhyRel, could help derive convenient alignments in situations where the control structure of the two programs isn't similar. Unconditional equivalence is itself a relational property, and we envision using existing infrastructure to reason about rewrites. Adding a sound rule for rewriting that is compatible with the encapsulation regime used in the logic is technically challenging and poses an interesting research problem.

## References

1. Banerjee, A., Nagasamudram, R., Nikouei, M., Naumann, D.A.: A relational program logic with data abstraction and dynamic framing (2019), available at `https://arxiv.org/abs/1910.14560`
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)
3. Banerjee, A., Naumann, D.A., Nikouei, M.: Relational logic with framing and hypotheses. In: IARCS. LIPIcs, vol. 65, pp. 11:1–11:16 (2016)
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)
5. Barthe, G., Crespo, J.M., Kunz, C.: Product programs and relational program logics. J. Logical and Algebraic Methods in Programming **85**(5), 847–859 (2016)
6. Beckert, B., Ulbrich, M.: Trends in relational program verification. In: Principled Software Development - Essays Dedicated to Arnd Poetzsch-Heffter. pp. 41–58 (2018)
7. Benton, N.: Simple relational correctness proofs for static analyses and program transformations. In: POPL. pp. 14–25. ACM (2004)
8. 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)
9. Churchill, B., Padon, O., Sharma, R., Aiken, A.: Semantic program alignment for equivalence checking. In: PLDI. p. 1027–1040 (2019)
10. Eilers, M., Müller, P., Hitz, S.: Modular product programs. ACM Trans. Program. Lang. Syst. **42**(1) (2019)
11. Filliâtre, J.C.: One Logic To Use Them All. In: CADE. Springer (2013)
12. Hoare, C.A.R.: Proofs of correctness of data representations. Acta Inf. **1**, 271–281 (1972)
13. Naumann, D.A.: Thirty-seven years of relational Hoare logic: remarks on its principles and history. In: ISOLA (2020)
14. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: CSL. pp. 1–19 (2001)