Ivy: Safety Verification by Interactive Generalization

Oded Padon

Tel Aviv University, Israel
odedp@mail.tau.ac.il

Kenneth L. McMillan

Microsoft Research, USA
kenmcmil@microsoft.com

Aurojit Panda

UC Berkeley, USA
apanda@cs.berkeley.edu

Mooly Sagiv

Tel Aviv University, Israel
msagiv@post.tau.ac.il

Sharon Shoham

Tel Aviv University, Israel
sharon.shoham@gmail.com

Abstract

Despite several decades of research, the problem of formal verification of infinite-state systems has resisted effective automation. We describe a system — Ivy — for *interactively* verifying safety of infinite-state systems. Ivy's key principle is that whenever verification fails, Ivy graphically displays a concrete *counterexample to induction*. The user then interactively guides generalization from this counterexample. This process continues until an inductive invariant is found. Ivy searches for universally quantified invariants, and uses a restricted modeling language. This ensures that all verification conditions can be checked algorithmically. All user interactions are performed using graphical models, easing the user's task. We describe our initial experience with verifying several distributed protocols.

Categories and Subject Descriptors   D.2.4 [*Software/Program Verification*]: Formal methods;   F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Invariants

Keywords   safety verification, invariant inference, counterexamples to induction, distributed systems

1. Introduction

Despite several decades of research, the problem of formal verification of systems with unboundedly many states has resisted effective automation. Although many techniques have been proposed, in practice they are either too narrow in scope or they make use of fragile heuristics that are highly sensitive to the encoding of the problem. In fact, most efforts towards verifying real-world systems use relatively little proof automation [8, 18, 20]. At best, they require a human user to annotate the system with an inductive invariant and use an automated decision procedure to check the resulting verification conditions.

Our hypothesis is that automated methods are difficult to apply in practice not primarily because they are unreliable, but rather because they are opaque. That is, they fail in ways that are difficult for a human user to understand and to remedy. A practical heuristic method, when it fails, should fail *visibly*, in the sense that the root cause of the failure is observable to the user, who can then provide an appropriate remedy. If this is true, the user can benefit from automated heuristics in the construction of the proof but does not reach a dead end in case heuristics fail.

Consider the problem of proving a safety property of a transition system. Most methods for this in one way or another construct and prove an inductive invariant. One way in which this process can fail is by failing to produce an inductive invariant. Typically, the root cause of this is failure to produce a useful generalization, resulting in an over-widening or divergence in an infinite sequence of abstraction refinements. Discovering this root cause in the complex chain of reasoning produced by the algorithm is often extremely difficult. To make such failures visible, we propose an interactive methodology that involves the user in the generalization process. We graphically visualize counterexamples to induction, and let the user guide the generalization with the help of automated procedures.

Another way the proof might fail is by failing to prove that the invariant is in fact inductive (for example, because of incompleteness of the prover). This can happen even when the invariant is provided manually. A typical root cause for this failure is that matching heuristics fail to produce a needed instantiation of a universal quantifier. Such failures of prover heuristics can be quite challenging for users to diagnose

and correct (and in fact the instability of heuristic matching posed significant difficulties for the proof effort of [8]). To eliminate this sort of invisible failure, we focus on universally quantified invariants and propose a restricted specification language that ensures that all verification conditions can be algorithmically checked.

We test these ideas by implementing them in a verification tool and applying it to a variety of infinite-state or parameterized systems. Although our correctness proofs were not obtained in a fully automated way, we find that automated generalization heuristics are still useful when applied with human guidance. Moreover, we find that the restrictions imposed by our specification language and by the use of universally quantified invariants are not an impediment to proving safety of parameterized distributed protocols.

Main Results The contributions of this paper are:
- A new methodology for safety verification of infinite-state systems via an interactive search for universally quantified inductive invariants. Our methodology combines user guidance with automated reasoning. To ensure decidability of the automated reasoning, the methodology requires that checking inductiveness of a universal invariant is decidable; and that checking if a universal property holds after a bounded number of transitions is decidable.
- A realization of our methodology using a new modeling language called RML (relational modeling language). RML is inspired by Alloy [16]. It represents program states using sets of first order relations. RML also allows functions to be used in a restricted form. Updates to relations and functions are restricted to be quantifier free. Non-deterministic statements in the style of Boogie and Dafny are supported. RML is designed to guarantee that verification conditions for every loop-free program fragment are expressible in an extension of the Bernays-Schönfinkel-Ramsey fragment of first-order logic, also known as EPR [26], for which checking satisfiability is decidable.
- A tool, called Ivy, that implements our new methodology for unbounded verification as a part of a verification framework. Ivy also allows model debugging via bounded verification. Using Ivy, we provide an initial evaluation of our methodology on some interesting distributed protocols modeled as RML programs. Ivy and the protocol models reported in this paper are publicly available [15].

2. Overview: Interactive Verification with Ivy

This section provides an informal overview of the verification procedure in Ivy.

Ivy's design philosophy Ivy is inspired by proof assistants such as Isabelle/HOL [23] and Coq [11] which engage the user in the verification process. Ivy also builds on success of tools such as Z3 [4] which can be very useful for bug finding, verification, and static analysis, and on automated

invariant inference techniques such as [17]. Ivy aims to balance between the predictability and visibility of proof assistants, and the automation of decision procedures and automated invariant inference techniques.

Compared to fully automated techniques, Ivy adopts a different philosophy which permits visible failures at the cost of more manual work from the users. Compared to proof assistants, Ivy provides the user with automated assistance, solving well-defined decidable problems. To obtain this, we use a restricted modeling language called RML. RML is restricted in a way which guarantees that the tasks performed automatically solve decidable problems. For example, RML does not allow arithmetic operations. However, RML is Turing-complete, and can be used to model many interesting infinite-state systems.

For systems modeled by RML programs, Ivy provides a verification framework which allows model debugging via bounded verification, as well as unbounded verification using our new methodology for interactively constructing universally quantified inductive invariants.

2.1 A Running Example: Leader Election

Figure 1 shows an RML program that models a standard protocol for leader election in a ring [3]. This example is used as a running example in this section. The protocol assumes a ring of unbounded size. Every node has a unique ID with a total order on the IDs. Thus, electing a leader can be done by a decentralized extrema-finding protocol. The protocol works by sending messages in the ring in one direction. Every node sends its own ID to its neighbor. A node forwards messages that contain an ID higher than its own ID. When a node receives a message with its own ID, it declares itself as a leader.

The RML program uses sorted variables, relations and a single function symbol to model the state of the protocol which evolves over time. Ivy allows only "stratified" function symbols (e.g., if there is a function mapping sort s_1 to sort s_2, there cannot be a function mapping s_2 to s_1). In the leader election example, IDs and nodes are modeled by sorts *id* and *node*, respectively. The function *id* maps each node to its ID. Since IDs are never mapped back to nodes by any function, *id* obeys the stratification requirement. The function *id* is uninterpreted, but the axiom *unique_ids* (line 11), which appears in Figure 2, constrains it to be injective (preventing two different nodes from having the same ID).

A binary relation *le* encodes a total order on IDs. The ring topology is represented by a ternary relation *btw* on nodes with suitable axiomatization that appears in Figure 2. $btw(x, y, z)$ holds for distinct elements x, y, z, if the shortest path in the ring from x to z goes through y (i.e., y is between x and z in the ring). *le* and *btw* are modeled as uninterpreted relations which are axiomatized in a sound and complete way using the universally quantified axioms *le_total_order* and *ring_topology* (lines 12 and 13), respectively. The unary relation *leader* holds for nodes which are identified as leaders.

```
1   sort node
2   sort id
3   function id : node → id
4   relation le : id,id
5   relation btw : node, node, node
6   relation leader: node
7   relation pnd: id, node
8   variable n : node
9   variable m : node
10  variable i : id
11  axiom unique_ids
12  axiom le_total_order
13  axiom ring_topology
14  assume ∀x.¬leader(x)
15  assume ∀x, y.¬pnd(x, y)
16  while * do {
17      assert ∀n₁, n₂.¬(leader(n₁)∧leader(n₂)∧n₁ ≠ n₂)
18      {
19        // send
20        n := *
21        m := *
22        assume next(n, m)
23        pnd.insert(id(n), m)
24      } | {
25        // receive
26        i := *
27        n := *
28        assume pnd(i, n)
29        pnd.remove(i, n)
30        if id(n) = i then {
31          leader.insert(n)
32        } else {
33          if le(id(n), i) then {
34            m := *
35            assume next(n, m)
36            pnd.insert(i, m)
37          }
38        }
39      }
40  }
```

Figure 1. An RML model of the leader election protocol. *unique_ids*, *le_total_order*, *ring_topology*, and *next(a, b)* denote a universally quantified formulas given in Figure 2.

$unique\_ids = \forall n_1, n_2. n_1 \neq n_2 \rightarrow id(n_1) \neq id(n_2)$
$le\_total\_order = \forall x.\, le(x, x)$ $\wedge$ $\forall x, y, z.\, le(x, y) \wedge le(y, z) \rightarrow le(x, z)$ $\wedge$ $\forall x, y.\, le(x, y) \wedge le(y, x) \rightarrow x = y$ $\wedge$ $\forall x, y.\, le(x, y) \vee le(y, x)$
$ring\_topology = \forall x, y, z.\, btw(x, y, z) \rightarrow btw(y, z, x)$ $\wedge$ $\forall w, x, y, z.\, btw(w, x, y) \wedge btw(w, y, z) \rightarrow btw(w, x, z)$ $\wedge$ $\forall w, x, y.\, btw(w, x, y) \rightarrow \neg btw(w, y, x)$ $\wedge$ $\forall w, x, y.\, \mathtt{distinct}(w, x, y) \rightarrow btw(w, x, y) \vee btw(w, y, x)$
$next(a, b) = \forall x.x \neq a \wedge x \neq b \rightarrow btw(a, b, x)$

Figure 2. Universally quantified formulas used in Figure 1. *unique_ids* expresses that no two nodes can have the same ID. *le_total_order* expresses that *le* is a reflexive total order. *ring_topology* expresses that `btw` represents a ring. *next(a, b)* expresses the fact that b is the immediate successor of a in the ring defined by *btw*.
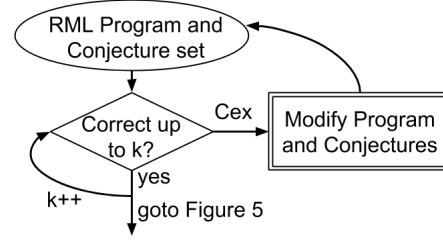

Figure 3. Flowchart of bounded verification.

The binary relation *pnd* between nodes and IDs denotes pending messages. The binary "macro" *next(a, b)* denotes the fact that node b is the immediate neighbor of node a in the ring. It is expressed by means using the *btw* relation by means of a universally quantified formula enforcing minimality. *next(a, b)* expresses the fact that b is the immediate neighbor of a in the ring.

In the initial state, no node is identified as a leader, and there are no pending messages. This is expressed by lines 14 and 15 respectively.

The core of the RML program is a non-deterministic loop. The loop starts by asserting that there is at most one leader (line 17). This assertion defines the safety property to verify. The loop body then contains a non-deterministic choice between two operations. The send operation (lines 19 to 23) sends a node's ID to its successor by inserting into the *pnd* relation. The receive operation (lines 25 to 38) handles an incoming message: it compares the ID in the message to the ID of the receiving node, and updates the *leader* and *pnd* relations according to the protocol.

Graphical visualization of states Ivy displays states of the protocol as graphs where the vertices represent elements, and edges represent relations and functions. As an example consider the state depicted in Figure 7 (a1). This state has two nodes and two IDs, represented by vertices of different shapes. Unary relations are displayed using vertex labels. For example, in Figure 7 (a1), node1 is labeled *leader*, and node2 is labeled ¬*leader*, denoting that the *leader* relation contains only node1. Binary relations such as *le* and *pnd* are displayed using directed edges. Higher-arity relations are displayed by means of their projections or derived relations. For example, the ternary relation *btw* is displayed by the derived binary relation *next* which captures a single ring edge. Functions such as *id* are displayed similarly to relations. The state depicted in Figure 7 (a1) contains two nodes, node1 and node2, such that the ID of node1 is lower (by *le*) than the ID of node2, the ID of node2 is pending at node2, and only node1 is contained in the *leader* relation. This state is clearly not reachable in the protocol, and the reason for displaying it will be explained in Section 2.3.

2.2 Bounded Verification

We aim for Ivy to be used by protocol designers to debug and verify their protocols. The first phase of the verification
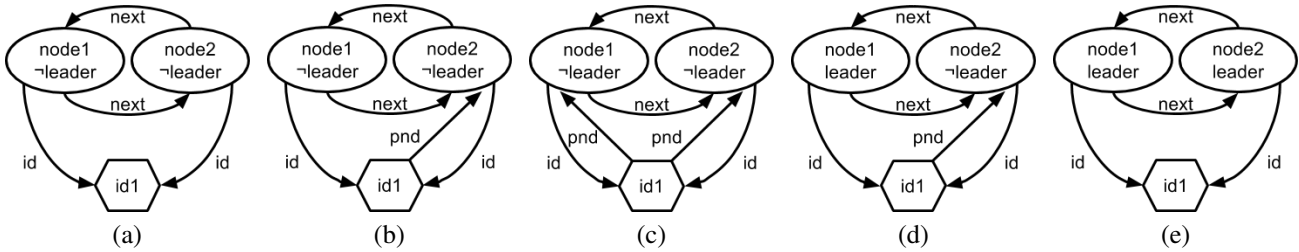
Figure 4. An error trace found by BMC for the leader protocol, when omitting the fact that node IDs are unique. (a) an initial state; (b) node1 sent a message to node2; (c) node2 sent a message to node1; (d) node1 processed a pending message and became leader; (e) node2 processed a pending message and became leader, there are now two leaders and the safety property is violated.

is debugging the protocol symbolically. Bounded model checking tools such as Alloy [16] can be very effective here. However, the restrictions on RML permit Ivy to implement a more powerful bounded verification procedure which does not a priori bound the size of the input configuration. Instead, Ivy tests if any k transitions (loop iterations) can lead to an assertion violation. In our experience this phase is very effective for bug finding since often the protocol and/or the desired properties are wrong.

For example, our initial modeling of the leader election protocol missed the *unique_ids* axiom (line 11). Bounded verification with a bound of 4 transitions resulted in the error trace depicted in Figure 3. In this trace, node1 identifies itself as a leader when it receives the message with the ID of node2 since they have the same ID (id1), and similarly for node2, leading to violation of the assertion. After adding the missing axiom, we ran Ivy with a bound of 10 transitions to debug the model, and did not get a counterexample trace. Notice again that Ivy does not restrict the size of the ring, only the number of loop iterations.

In our experience, protocols can be verified for about 10 transitions in a few minutes. Once bounded verification does not find more bugs, the user can prove unbounded correctness by searching for an inductive invariant.

2.3 Interactive Search for Universally Quantified Inductive Invariants

The second phase of the verification is to find a universally quantified inductive invariant that proves that the system is correct for any number of transitions. This phase requires more user effort but enables ultimate safety verification.

We say that an invariant I is *inductive* for an RML program if: (i) All initial states of the program satisfy I (**initiation**). (ii) Every state satisfying I also satisfies the desired safety properties (**safety**). (iii) I is closed under the transitions of the program, i.e., executing the loop body from any arbitrary program state satisfying I results in a new program state which also satisfies I (**consecution**).

If the user has a universally quantified inductive invariant in mind, Ivy can automatically check if it is indeed an inductive invariant. Due to the restrictions of RML, this
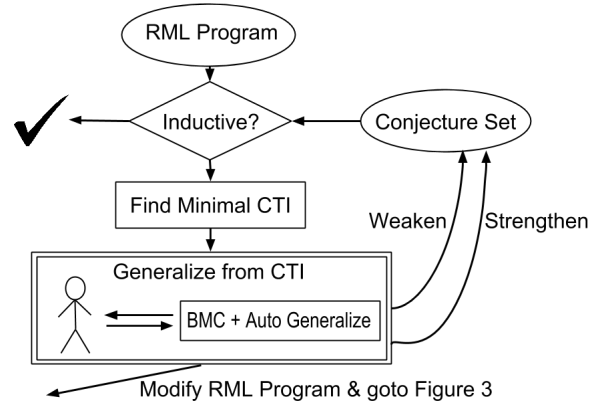

Figure 5. Flowchart of the interactive search for an inductive invariant.

check is guaranteed to terminate with either a proof showing that the invariant is inductive or a finite counterexample which can be depicted graphically and presented to the user. We refer to such a counterexample as a *counterexample to induction (CTI)*. A CTI does not necessarily imply that the safety property is violated — only that I is not an inductive invariant. Coming up with inductive invariants for infinite-state systems is very difficult. Therefore, Ivy supports an interactive procedure for gradually obtaining an inductive invariant or deciding that the RML program or the safety property need to be revised.

The search for an inductive invariant starts with a (possibly empty) set of universally quantified conjectures, and advances based on CTIs according to the procedure described in Figure 5. When a CTI is found it is displayed graphically, and the user has 3 options: 1) The user understands that there is a bug in the model or safety property, in which case the user revises the RML program and starts over in Figure 3. Note that in this case, the user may choose to retain some conjectures reflecting gained knowledge of the expected protocol behavior. 2) The user understands that one of the conjectures is wrong, in which case the user removes it from the conjecture set, weakening the candidate invariant. 3) The user judges that the CTI is not reachable. This means that the invariant needs to be strengthened by adding a conjecture.

The new conjecture should eliminate the CTI, and should generalize from it. This is the most creative task, and our approach for it is explained below.

Graphical visualization of conjectures To allow the user to examine and modify possible conjectures, Ivy provides a graphical visualization of universally quantified conjectures. Such a conjecture asserts that some sub-configuration of the system is not present in any reachable state. That is, any state of the system that contains this sub-configuration is not reachable. Ivy graphically depicts such conjectures by displaying the forbidden sub-configuration. Sub-configurations are visualized similarly to the way states are displayed, but with a different semantics.

As an example consider the conjecture depicted in Figure 7 (b). The visualization shows two nodes and their distinct IDs; node1 is shown to be a leader, while node2 is not a leader. Furthermore, the ID of node1 is lower (by *le*) than the ID of node2. Note that no pending messages appear (no *pnd* edges), and there is also no information about the topology (no *next* or *btw* edges). Viewed as a conjecture, this graph asserts that in any reachable state, there cannot be two nodes such that the node with the lower ID is a leader and the node with the higher ID is not a leader. Thus, this conjecture excludes infinitely many states with any number of nodes above 2 and any number of pending messages. It excludes all states that contain any two nodes such that the node with the lower ID is a leader and the node with the higher ID is not a leader.

Figure 7 (c) depicts an even stronger (more general) conjecture: unlike Figure 7 (b), node2 is not labeled with ¬*leader* nor with *leader*. This means that the conjecture in Figure 7 (c) excludes all the states that contain two nodes such that the node with the lower ID is a leader, regardless of whether the other node is a leader or not.

Obtaining helpful CTIs Since we rely on the user to guide the generalization, it is critical to display a CTI that is easy to understand and indicative of the proof failure. Therefore, Ivy searches for "minimal" CTIs. `Find Minimal CTI` automatically obtains a minimal CTI based on user provided minimization parameters. Examples include minimizing the number of elements, and minimizing certain relations (e.g. minimizing the *pnd* relation).

Interactive generalization from CTIs When a CTI represents an unreachable state, we should strengthen the invariant by adding a new conjecture to eliminate the CTI. One possible universally quantified conjecture is the one which excludes all states that contain the concrete CTI as a sub-configuration (formally, a substructure) [17]. However, this conjecture may be too specific, as the CTI contains many features that are not relevant to the failure. This is where generalization is required, or otherwise we may end up in a diverging refinement loop, always strengthening the invariant with more conjectures that are all too specific.

C_0	$\forall n_1, n_2. \neg(leader(n_1) \wedge leader(n_2) \wedge n_1 \neq n_2)$
C_1	$\forall n_1, n_2. \neg(n_1 \neq n_2 \wedge leader(n_1) \wedge le(id(n_1), id(n_2)))$
C_2	$\forall n_1, n_2. \neg(n_1 \neq n_2 \wedge pnd(id(n_1), n_1) \wedge le(id(n_1), id(n_2)))$
C_3	$\forall n_1, n_2, n_3. \neg(btw(n_1, n_2, n_3) \wedge pnd(id(n_2), n_1) \wedge$ $le(id(n_2), id(n_3)))$

Figure 6. The conjectures found using Ivy for the leader election protocol. C_0 is the safety property, and the remaining conjectures (C_1 - C_3) were produced interactively. $C_0 \wedge C_1 \wedge C_2 \wedge C_3$ is an inductive invariant for the protocol.

This is also where Ivy benefits from user intuition beyond automatic tools, as it asks the user to guide generalization. Ivy presents the user with a concrete CTI, and lets the user eliminate some of the features of the CTI that the user judges to be irrelevant. This already defines a generalization of the CTI that excludes more states.

Next, the `BMC + Auto Generalize` procedure, applies bounded verification (with a user-specified bound) to check the user's suggestion and generalize further. If the test fails, it means that the user's generalized conjecture is violated in a reachable state, and a concrete counterexample trace is displayed to let the user diagnose the problem. If the test succeeds (i.e., the bounded verification formula is unsatisfiable), Ivy automatically suggests a stronger generalization, based on a minimal UNSAT core. The user then decides whether to accept the suggested conjecture and add it to the invariant, or to change the parameters in order to obtain a different suggestion.

Next, we walk through this process for the leader election protocol, demonstrating the different stages, until we obtain an inductive invariant that proves the protocol's safety.

Illustration using the leader election protocol Figure 6 summarizes the 3 iterations Ivy required to find an inductive invariant for the leader election protocol. The initial set of conjectures contains only C_0, which is derived from the assertion in Figure 1 line 17.

In the first iteration, since C_0 alone is not inductive, Ivy applies `Find Minimal CTI`. This results in the CTI depicted in Figure 7 (a1). Figure 7 (a2) depicts a successor state of (a1) reached after node2 receives the pending message with its ID. The state (a1) satisfies C_0, whereas (a2) violates it, making (a1) a CTI. After examining this CTI, the user judges that the state (a1) is unreachable, with the intuitive explanation that node1 identifies itself as a leader despite the fact that node2 has a higher ID. Thus, the user generalizes away the irrelevant information, which includes *pnd* and the ring topology, resulting in the generalization depicted in Figure 7 (b).

Next, the user applies `BMC + Auto Generalize` with bound 3 to this generalization. The BMC test succeeds, and Ivy suggests the generalization in Figure 7 (c), where the information that node2 is not a leader is also abstracted away.
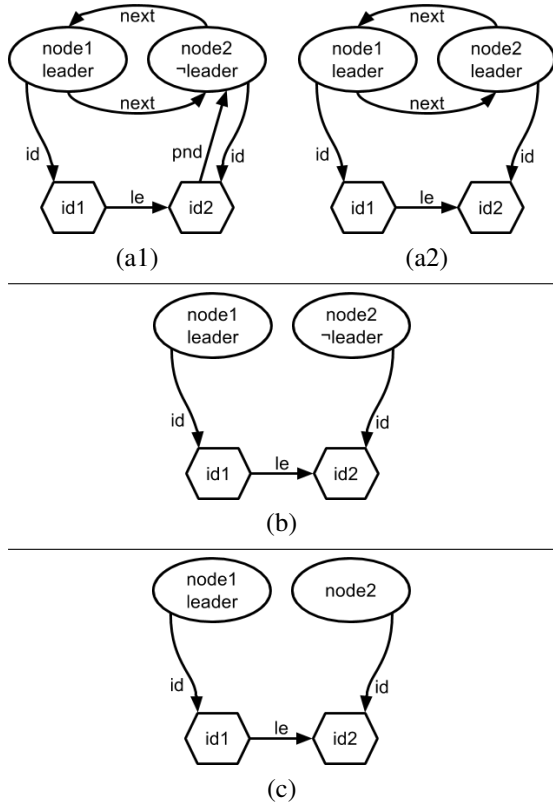
Figure 7. The 1st CTI generalization step for the leader protocol, leading to C_1. (a1) The CTI state that has one leader, but makes a transition to a state (a2) with two leaders. The root cause is that a node with non-maximal ID is a leader. (b) A generalization created by the user by removing the topology information and the *pnd* relation (c) Further generalization obtained by `BMC + Auto Generalize`, which removed the fact that node2 is not a leader.

The user approves this generalization, which corresponds to conjecture C_1 shown in Figure 6, so C_1 is added to the set of conjectures.

If the user had used bound 2 instead of 3 when applying `BMC + Auto Generalize`, then Ivy would have suggested a stronger generalization that also abstracts the ID information, and states that if there are two distinct nodes, none of them can be a leader. This conjecture is bogus, but it is true for up to 2 loop iterations (since with 2 nodes, a node can only become a leader after a send action followed by 2 receive actions). It is therefore the role of the user to select and adjust the bound for automatic generalization, and to identify bogus generalizations when they are encountered.

After adding the correct conjecture C_1, Ivy displays the CTI depicted in Figure 8 (a1) with its successor state (a2). Note that (a2) does not violate the safety property, but it violates C_1 that was added to the invariant, since a node with a non-maximal ID becomes a leader. The user examines (a1) and concludes that it is not reachable, since it has a pending
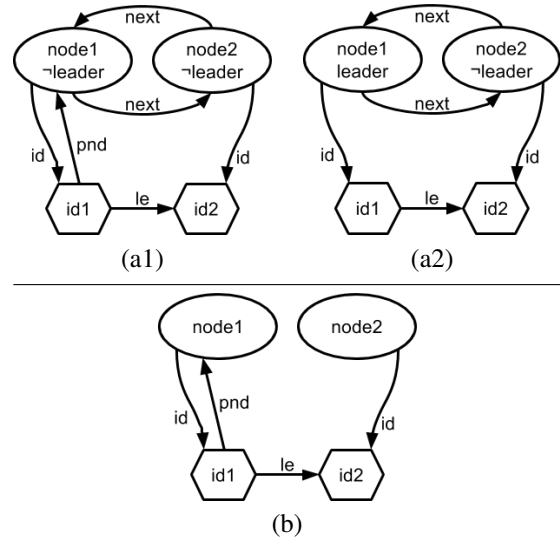
Figure 8. The 2nd CTI generalization step for the leader protocol, leading to C_2. (a1) The CTI state and its successor (a2) violating C_1. The root cause is that a node with non-maximal ID has a pending message with its own ID. (b) A generalization created by the user by removing the topology information and the leader relation. This generalization was validated but not further generalized by `BMC + Auto Generalize`.

message to node1 with its own ID, despite the fact that node2 has a higher ID. Here, the user again realizes that the ring topology is irrelevant and abstracts it away. The user also abstracts away the *leader* information. On the other hand, the user keeps the *pnd* information, in accordance with the intuitive explanation of why the CTI is not reachable. The resulting user-defined generalization is depicted in Figure 8 (b). `BMC + Auto Generalize` with bound 3 validates this generalization for 3 transitions, but does not manage to generalize any further. Thus, the generalization is converted to C_2 in Figure 6 which is added to the invariant, and the process continues.

Finally, Figure 9 (a1) and (a2) depicts a CTI that leads to a violation of C_2. This CTI contains three nodes, with a pending message that appears to bypass a node with a higher ID. This time, the user does not abstract away the topology since it is critical to the reason the CTI is not reachable. The user only abstracts the *leader* information, which leads to the generalization depicted in Figure 9 (b). Note that in the generalized conjecture we no longer consider the *next* relation, but rather the *btw* relation. This expresses the fact that, as opposed to the concrete CTI, the conjecture generalizes from the specific topology of a ring with exactly 3 nodes, to a ring that contains it as a sub-configuration, i.e. a ring with at least 3 nodes that are not necessarily immediate neighbors of each other. We do require that the three nodes are ordered in such a way that node2 is between node1 and
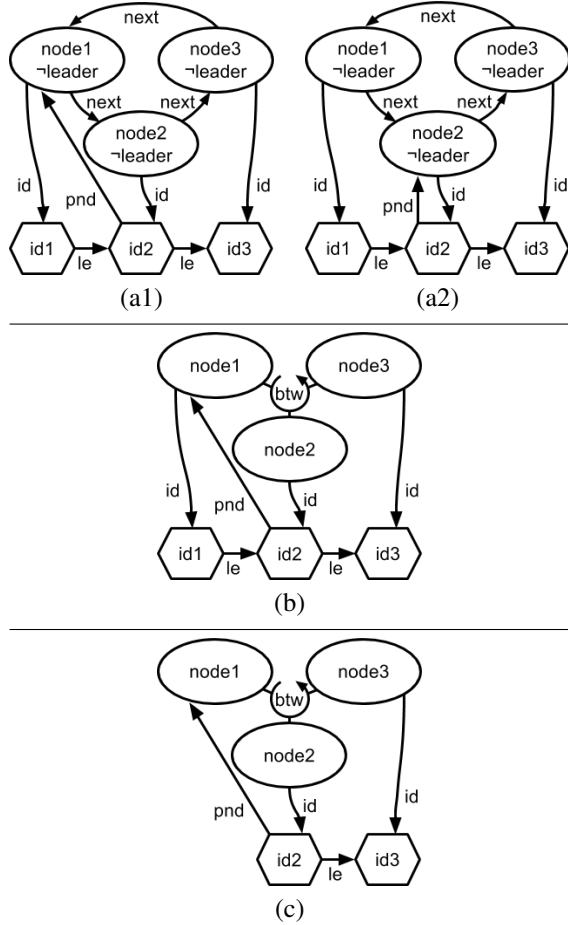
Figure 9. The 3rd CTI generalization step for the leader protocol, leading to C_3. (a1) The CTI state and its successor (a2) which violates C_2. The root cause is that node1 has a pending message with the ID of node2, even though node3 is on the path from node2 to node1 and has an ID higher than node2's ID (equivalently, node2 is between node1 and node3 and has a lower ID than node3's ID). (b) A generalization created by the user by removing the leader relation. The generalization does not contain *next*, only *btw*. (c) Further generalization obtained by `BMC + Auto Generalize`, which eliminated id1.

node2 in the ring (equivalently, node3 is between node2 and node1).

Applying `BMC + Auto Generalize` with a bound of 3 to Figure 9 (b) confirms this conjecture, and automatically abstracts away the ID of node1, which results in the conjecture depicted in Figure 9 (c), which corresponds to C_3 in Figure 6. The user adds this conjecture to the invariant. After adding C_3, Ivy reports that $I = C_0 \land C_1 \land C_2 \land C_3$ is an inductive invariant for the leader election protocol.

3. RML: Relational Modeling Language with Effectively Propositional Reasoning

In this section we define a simple modeling language, called Relational Modeling Language (RML). RML is Turing-complete, and suitable for modeling infinite-state systems. RML is restricted in a way that ensures that checking verification conditions for RML programs is decidable, which enables our interactive methodology for safety verification. We start with RML's syntax and informal semantics. We then define a weakest precondition operator for RML, which is used in the verification conditions.

3.1 RML Syntax and Informal Semantics

Figures 10 and 11 show the abstract syntax of RML. RML imposes two main programming limitations: (i) the only data structures are uninterpreted finite relations and stratified functions, and (ii) program conditions and update formulas have restricted quantifier structure.

RML further restricts programs to consist of a single non-deterministic loop, with possible initialization and finalization commands. Thus, RML commands are loop-free. While this restriction simplifies the presentation of our approach, note that it does not reduce RML's expressive power as nested loops can always be converted to a flat loop.

Declarations and states The declarations of an RML program define a set of sorts $\mathcal{S}$, a set of sorted relations $\mathcal{R}$, a set of sorted functions $\mathcal{F}$, a set of sorted program variables $\mathcal{V}$, and a set of axioms $\mathcal{A}$ in the form of (closed) $\exists^*\forall^*$-formulas.

A state of an RML program associates a finite set of elements (domain) with each sort, and defines valuations (interpretations) for all relations, functions, and variables, such that all the axioms are satisfied. The values of relations and functions can be viewed as finite tables (or finite sets of tuples). The values of program variables, relations and functions are all mutable by the program.

Note that while every program state can be described by finite tables, there is no bound on the size of the tables, and thus an RML program has infinitely many states and can model infinite state systems.

For example, a network with an unbounded number of nodes can be modeled with sorts for nodes and messages, and a relation that keeps track of pending messages in the network. Functions can be used to relate each node to a unique ID and other node specific data, and to relate each message to its source, destination, and other message fields. While in any given state the number of nodes and the number of pending messages is finite, there is no bound on the possible number of nodes or pending messages.

Function stratification The set of functions $\mathcal{F}$ is required to be *stratified*. This means that the sorts of the program can be ordered by a total order $<$ such that if $\mathcal{F}$ contains a function $\mathbf{f} : \mathbf{s}_1, \ldots, \mathbf{s}_n \to \mathbf{s}'$ then $\mathbf{s}' < \mathbf{s}_i$ for every $1 \leq i \leq n$. For example, if $\mathcal{F}$ contains a function from

$$\begin{array}{lll}
\langle rml\rangle & ::= & \langle decls\rangle \; ; \; \langle cmd\rangle \; ; \; \texttt{while} * \texttt{do} \; \langle cmd\rangle \; ; \; \langle cmd\rangle
\end{array}$$

$\langle decls\rangle$	$::= \; \epsilon \;\; \mid \;\; \langle decls\rangle; \langle decls\rangle$	
	$\mid$ $\;\;$ $\texttt{sort}\;\mathbf{s}$	
	$\mid$ $\;\;$ $\texttt{relation}\;\mathbf{r}:\overline{\mathbf{s}}$	
	$\mid$ $\;\;$ $\texttt{function}\;\mathbf{f}:\overline{\mathbf{s}}\to\mathbf{s}$	
	$\mid$ $\;\;$ $\texttt{variable}\;\mathbf{v}:\mathbf{s}$	
	$\mid$ $\;\;$ $\texttt{axiom}\;\varphi_{\mathrm{EA}}$	

$\langle cmd\rangle$	$::= \; \texttt{skip}$	do nothing
	$\mid$ $\;\;$ $\texttt{abort}$	terminate abnormally
	$\mid$ $\;\;$ $\mathbf{r}\,(\overline{x}) := \varphi_{\mathrm{QF}}\,(\overline{x})$	quantifier-free update of relation $\mathbf{r}$
	$\mid$ $\;\;$ $\mathbf{f}\,(\overline{x}) := t\,(\overline{x})$	update of function $\mathbf{f}$ to term $t\,(\overline{x})$
	$\mid$ $\;\;$ $\mathbf{v} := *$	havoc of variable $\mathbf{v}$
	$\mid$ $\;\;$ $\texttt{assume}\;\varphi_{\mathrm{EA}}$	assume $\exists^*\forall^*$ formula holds
	$\mid$ $\;\;$ $\langle cmd\rangle \; ; \; \langle cmd\rangle$	sequential composition
	$\mid$ $\;\;$ $\langle cmd\rangle \mid \langle cmd\rangle$	non-deterministic choice

Figure 10. Syntax of RML. $\mathbf{s}$ denotes a sort identifier and $\overline{\mathbf{s}}$ denotes a vector of sort identifiers separated by commas. $\mathbf{r}$ denotes a relation identifier. $\mathbf{f}$ denotes a function identifier. $\mathbf{v}$ denotes an identifier of a zero-arity function. $\overline{x}$ denotes a vector of logical variables. $t\,(\overline{x})$ denotes a term with free logical variables $\overline{x}$ and $\varphi_{\mathrm{QF}}\,(\overline{x})$ denotes a quantifier-free formula with free logical variables $\overline{x}$. φ_{EA} denotes a closed formula with quantifier prefix $\exists^*\forall^*$. The syntax of terms and formulas is given in Figure 11.

messages to nodes (e.g. the message source), then $\mathcal{F}$ cannot contain a function from nodes to messages. This restriction on functions is essential for the decidability properties of RML (see Section 3.3).

Commands Each command investigates and potentially updates the state of the program. The semantics of $\texttt{skip}$ and $\texttt{abort}$ are standard. The command $\mathbf{r}(x_1, \ldots, x_n) := \varphi_{\mathrm{QF}}(x_1, \ldots, x_n)$ is used to update the n-ary relation $\mathbf{r}$ to the set of all n-tuples that satisfy the quantifier-free formula φ_{QF}. For example, $\mathbf{r}(x_1, x_2) := (x_1 = x_2)$ updates the binary relation $\mathbf{r}$ to the identity relation; $\mathbf{r}(x_1, x_2) := \mathbf{r}(x_2, x_1)$ updates $\mathbf{r}$ to its inverse relation; $\mathbf{r}_1(x) := \mathbf{r}_2(x, \mathbf{v})$ updates $\mathbf{r}_1$ to the set of all elements that are related by $\mathbf{r}_2$ to the current value (interpretation) of program variable $\mathbf{v}$.

The command $\mathbf{f}(x_1, \ldots, x_n) := t(x_1, \ldots, x_n)$ is used to update the n-ary function $\mathbf{f}$ to match every n-tuple of elements to the element given by the term t. Note that while relations are updated to quantifier-free formulas, functions are updated to terms. For example, $\mathbf{f}(x) := x$ updates the function $\mathbf{f}$ to the identity function; $\mathbf{f}(x_1, x_2) := \mathbf{f}(x_2, x_1)$ updates $\mathbf{f}$ to its transpose; $\mathbf{f}(x) := \mathrm{ite}(\mathbf{r}(x), x, \mathbf{f}(x))$ updates $\mathbf{f}$ to be the identity function for all elements in $\mathbf{r}$, and leaves it unchanged for all elements not in $\mathbf{r}$.

The havoc command $\mathbf{v} := *$ performs a non-deterministic assignment to $\mathbf{v}$. The $\texttt{assume}$ command is used to restrict the executions of the program to those that satisfy the given

$\langle t\rangle$	$::= \; x$	logical variable x
	$\mid$ $\;\;$ $\mathbf{v}$	program variable $\mathbf{v}$
	$\mid$ $\;\;$ $\mathbf{f}\,(\langle t\rangle, \ldots, \langle t\rangle)$	application of function $\mathbf{f}$
	$\mid$ $\;\;$ $\mathrm{ite}\,(\varphi_{\mathrm{QF}}, \langle t\rangle, \langle t\rangle)$	if-then-else term

$\langle \varphi_{\mathrm{QF}}\rangle$	$::= \; \mathbf{r}\,(\langle t\rangle, \ldots, \langle t\rangle)$	membership in relation $\mathbf{r}$
	$\mid$ $\;\;$ $\langle t\rangle = \langle t\rangle$	equality between terms
	$\mid$ $\;\;$ $\langle \varphi_{\mathrm{QF}}\rangle \wedge \langle \varphi_{\mathrm{QF}}\rangle \;\mid\; \langle \varphi_{\mathrm{QF}}\rangle \vee \langle \varphi_{\mathrm{QF}}\rangle \;\mid\; \neg\langle \varphi_{\mathrm{QF}}\rangle$	

$\langle \varphi_{\mathrm{AF}}\rangle$	$::= \; \exists x_1, \ldots, x_n \; \langle \varphi_{\mathrm{QF}}\rangle \;\mid\; \forall x_1, \ldots, x_n \; \langle \varphi_{\mathrm{QF}}\rangle$	
	$\mid$ $\;\;$ $\langle \varphi_{\mathrm{AF}}\rangle \wedge \langle \varphi_{\mathrm{AF}}\rangle \;\mid\; \langle \varphi_{\mathrm{AF}}\rangle \vee \langle \varphi_{\mathrm{AF}}\rangle \;\mid\; \neg\langle \varphi_{\mathrm{AF}}\rangle$	

$\langle \varphi_{\mathrm{EA}}\rangle$	$::= \; \exists x_1, \ldots, x_n \; \forall x_{n+1}, \ldots, x_{n+m} \; \langle \varphi_{\mathrm{QF}}\rangle$
$\langle \varphi_{\mathrm{AE}}\rangle$	$::= \; \forall x_1, \ldots, x_n \; \exists x_{n+1}, \ldots, x_{n+m} \; \langle \varphi_{\mathrm{QF}}\rangle$

Figure 11. Syntax of terms and formulas. Formulas in $\langle \varphi_{\mathrm{AF}}\rangle$ (alternation-free formulas), $\langle \varphi_{\mathrm{EA}}\rangle$ ($\exists^*\forall^*$-formulas), and $\langle \varphi_{\mathrm{AE}}\rangle$ ($\forall^*\exists^*$-formulas) are assumed to be closed (without free logical variables). Note that altrenation-free formulas are closed under negation, and negating an $\exists^*\forall^*$-formula results in a $\forall^*\exists^*$-formula and vice versa (after converting to prenex normal form).

Syntactic Sugar	RML Command
$\texttt{assert}\;\varphi_{\mathrm{AE}}$	$\{\texttt{assume}\;\neg\varphi_{\mathrm{AE}}\;;\;\texttt{abort}\}$ $\mid\;\texttt{skip}$
$\texttt{if}\;\varphi_{\mathrm{AF}}\;\texttt{then}\;C_1\;\texttt{else}\;C_2$	$\{\texttt{assume}\;\varphi_{\mathrm{AF}}\;;\;C_1\}\mid$ $\{\texttt{assume}\;\neg\varphi_{\mathrm{AF}}\;;\;C_2\}$
$\mathbf{r}.\texttt{insert}\,(\overline{x}\mid\varphi_{QF}\,(\overline{x}))$	$\mathbf{r}\,(\overline{x}) := \mathbf{r}\,(\overline{x}) \vee \varphi_{QF}\,(\overline{x})$
$\mathbf{r}.\texttt{remove}\,(\overline{x}\mid\varphi_{QF}\,(\overline{x}))$	$\mathbf{r}\,(\overline{x}) := \mathbf{r}\,(\overline{x}) \wedge \neg\varphi_{QF}\,(\overline{x})$
$\mathbf{r}.\texttt{insert}\,(\overline{t})$	$\mathbf{r}\,(\overline{x}) := \mathbf{r}\,(\overline{x}) \vee (\overline{x} = \overline{t})$
$\mathbf{r}.\texttt{remove}\,(\overline{t})$	$\mathbf{r}\,(\overline{x}) := \mathbf{r}\,(\overline{x}) \wedge \neg(\overline{x} = \overline{t})$
$\mathbf{f}\,(\overline{t}) := t$	$\mathbf{f}\,(\overline{x}) := \mathrm{ite}\,(\overline{x} = \overline{t}, t, \mathbf{f}\,(\overline{x}))$

Figure 12. Syntactic sugars for RML. φ_{AE} denotes a formula with $\forall^*\exists^*$ prefix. φ_{AF} denotes an alternation-free formula. $\mathbf{r}$ denotes an n-ary relation, $\mathbf{f}$ denotes an n-ary function, $\overline{x}$ denotes a vector of n logical variables, and $\overline{t}$ denotes a vector of n closed terms.

(closed) $\exists^*\forall^*$-formula. Sequential composition and non-deterministic choice are defined in the usual way.

The commands given in Figure 10 are the core of RML. Figure 12 provides several useful syntactic sugars for RML, including an $\texttt{assert}$ command, an $\texttt{if-then-else}$ command, and convenient update commands for relations and functions.

Executions and safety An RML program has the form $decls\;;\;C_{\texttt{init}}\;;\;\texttt{while}\;*\;\texttt{do}\;C_{\texttt{body}}\;;\;C_{\texttt{final}}$. Execution traces of an RML program are defined as a sequence of states that correspond to executing $C_{\texttt{init}}$ (from any state satisfying the axioms $\mathcal{A}$), then executing $C_{\texttt{body}}$ any number of times (including zero), and then executing $C_{\texttt{final}}$. An $\texttt{abort}$ command aborts the execution and terminates the trace. A

trace that leads to the execution of an abort command is called an *error trace*. An RML program is *safe* if it has no error traces.

Turing-completeness   To see that RML is Turing-complete, we can encode a (Minsky) counter machine in RML. Each counter c_i can be encoded with a unary relation r_i. The value of counter c_i is the number of elements in r_i. Testing for zero, incrementing, and decrementing counters can all be easily expressed by RML commands.

3.2   Axiomatic Semantics

We now provide a formal semantics for RML by defining a weakest precondition operator for RML commands with respect to assertions expressed in sorted first-order logic. We start with a formal definition of program states as structures of sorted first-order logic, and program assertions as formulas in sorted first-order logic.

States   Recall that an RML program declares a set of program variables $\mathcal{V}$, relations $\mathcal{R}$, and functions $\mathcal{F}$. We define a sorted first-order vocabulary Σ, that contains a relation symbol for every relation in $\mathcal{R}$, a function symbol for every function in $\mathcal{F}$, and a nullary function symbol for every variable in $\mathcal{V}$, all with appropriate sorts. A state of the program is given by a sorted first-order structure over Σ, defined as follows.

Definition 1 (Structures). Given a vocabulary Σ, a *structure* of Σ is a pair $s = (D, \mathcal{I})$, where D is a *finite* sorted domain, and $\mathcal{I}$ is an interpretation function, mapping each symbol of Σ to its meaning in s. $\mathcal{I}$ associates each k-ary relation symbol $r \in \Sigma$ with a function $\mathcal{I}(r) : D^k \rightarrow \{0,1\}$, and associates each k-ary function symbol $f \in \Sigma$ with a function $\mathcal{I}(f) : D^{k+1} \rightarrow \{0,1\}$ such that for any $x_1, \ldots, x_k \in D$, $\mathcal{I}(f)(x_1, \ldots, x_k, y) = 1$ for exactly one element $y \in D$ [1]. $\mathcal{I}$ also obeys the sort restrictions.

The states of an RML program are (finite) structures of Σ that satisfy all the axioms $\mathcal{A}$ declared by the program.

Assertions   Assertions on program states are specified by closed formulas (i.e., without free logical variables) in sorted first-order logic over Σ (see Figure 11). In the sequel, we use assertions and formulas interchangeably, and they are always assumed to be closed. A state satisfies an assertion if it satisfies it in the usual semantics of sorted first-order logic.

Remark 3.1.   The reader should be careful not to confuse program variables (modeled as nullary function symbols in Σ) with logical variables used in first-order formulas.

Weakest precondition of RML commands   Figure 13 presents the definition of a *weakest precondition* operator for RML, denoted wp. The weakest precondition [5] of a command C with respect to an assertion Q, denoted $wp(C, Q)$, is

$$wp\,(\texttt{skip}, Q) = Q$$
$$wp\,(\texttt{abort}, Q) = \textit{false}$$
$$wp\,(\mathbf{r}\,(\overline{x}) := \varphi_{\text{QF}}\,(\overline{x})\,, Q) = (\mathcal{A} \rightarrow Q)\,[\varphi_{\text{QF}}\,(\overline{s})\ /\ \mathbf{r}\,(\overline{s})]$$
$$wp\,(\mathbf{f}\,(\overline{x}) := t\,(\overline{x})\,, Q) = (\mathcal{A} \rightarrow Q)\,[t\,(\overline{s})\ /\ \mathbf{f}\,(\overline{s})]$$
$$wp\,(\mathbf{v} := *, Q) = \forall x.\,(\mathcal{A} \rightarrow Q)\,[x\ /\ \mathbf{v}]$$
$$wp\,(\texttt{assume}\ \varphi_{\text{EA}}, Q) = \varphi_{\text{EA}} \rightarrow Q$$
$$wp\,(C_1\,;\ C_2, Q) = wp\,(C_1, wp\,(C_2, Q))$$
$$wp\,(C_1\,|\,C_2, Q) = wp\,(C_1, Q) \wedge wp\,(C_2, Q)$$

Figure 13. Rules for wp. $\varphi\,[\beta\,/\,\alpha]$ denotes φ with occurrences of α substituted by β. $\overline{s}$ denotes a vector of terms.

an assertion Q' such that every execution of C starting from a state that satisfies Q' leads to a state that satisfies Q. Further, $wp(C, Q)$ is the weakest such assertion. Namely, $Q' \Rightarrow wp(C, Q)$ for every Q' as above.

The rules for wp of skip and abort are standard, as are the rules for assume, sequential composition and nondeterministic choice. The rules for updates of relations and functions and for havoc are instances of Hoare's assignment rule [10], applied to the setting of RML and adjusted for the fact that state mutations are restricted by the axioms $\mathcal{A}$.

Safety   Recall that an RML program is safe if it has no execution trace that leads to an abort command, and that an execution trace of an RML program consists of executing $C_{\texttt{init}}$ from any state satisfying the axioms $\mathcal{A}$, then executing $C_{\texttt{body}}$ any number of times (including zero), and then executing $C_{\texttt{final}}$. To formalize safety using wp, denote $C_{\texttt{body}}^0 = \texttt{skip}$ and $C_{\texttt{body}}^{k+1} = C_{\texttt{body}}^k\,;\ C_{\texttt{body}}$ for any $k \geq 0$. Thus, $C_{\texttt{body}}^k$ is $C_{\texttt{body}}$ repeated sequentially k times. An RML program is safe iff for any $k \geq 0$ we have

$$\mathcal{A} \Rightarrow wp\,\big(C_{\texttt{init}}\,;\ C_{\texttt{body}}^k\,;\ C_{\texttt{final}}, \textit{true}\big) \quad (k\text{-safety})$$
$$(1)$$

where $\mathcal{A}$ is the set of axioms of the program.

Inductive invariants   The safety of an RML program can be established by providing a formula I, such that the following conditions hold:

$$
\begin{array}{rcll}
\mathcal{A} & \Rightarrow & wp\,(C_{\texttt{init}}, I) & \text{(initiation)} \\
\mathcal{A} \wedge I & \Rightarrow & wp\,(C_{\texttt{final}}, \textit{true}) & \text{(safety)} \\
\mathcal{A} \wedge I & \Rightarrow & wp\,(C_{\texttt{body}}, I) & \text{(consecution)}
\end{array}
\qquad (2)
$$

A formula that satisfies these conditions is called an *inductive invariant* for the RML program.

3.3   RML Decidability Properties

We now show how the restrictions to quantifier-free updates and $\exists^*\forall^*$ assume's, combined with the requirement that function symbols are stratified, lead to decidability of verification conditions for RML programs.

A key property of the wp operator (Figure 13) is that $\forall^*\exists^*$-formulas are closed under wp:

---

[1] For uniformity of the presentation of our approach, we treat functions of arity k as "special" relations of arity $k+1$ that relate each k-tuple to exactly one element.

Lemma 3.2. *Let C be an RML command. If Q is a $\forall^*\exists^*$-formula, then so is the prenex normal form of $wp(C, Q)$.*

Note that this property depends both on the restriction of updates to quantifier-free formulas, and on the restriction of `assume`'s and axioms to $\exists^*\forall^*$-formulas.

EPR The effectively-propositional (EPR) fragment of first-order logic, also known as the Bernays-Schönfinkel-Ramsey class is restricted to relational first-order formulas (i.e., formulas over a vocabulary that contains constant symbols and relation symbols but no function symbols) with a quantifier prefix $\exists^*\forall^*$. Satisfiability of EPR formulas is decidable [21]. Moreover, formulas in this fragment enjoy the *finite model property*, meaning that a satisfiable formula is guaranteed to have a finite model. The size of this model is bounded by the total number of existential quantifiers and constants in the formula. The reason for this is that given an $\exists^*\forall^*$-formula, we can obtain an equi-satisfiable quantifier-free formula by replacing the existentially quantified variables by Skolem constants, and then instantiating the universal quantifiers for all constants and Skolem constants.

While EPR does not allow any function symbols, it can be easily extended to allow stratified function symbols while maintaining both the finite model property and the decidability of the satisfiability problem (though the models may be larger). The reason for this is that, for a finite set of constant and Skolem constant symbols, stratified function symbols can only generate a finite number of ground terms (for a similar procedure, see [13]).

Decidability of checking RML verification conditions A common task that arises when verifying an RML program is to check, for a given RML command C and formulas P and Q, whether or not $P \Rightarrow wp(C, Q)$, as in Equations (1) and (2). The following theorem shows that if P and Q have the right quantifier structure, then this check is decidable.

Theorem 3.3. *Let C be an RML command, P be an $\exists^*\forall^*$-formula, and Q be a $\forall^*\exists^*$-formula. Then, checking if $P \Rightarrow wp(C, Q)$ is decidable. Furthermore, if $P \not\Rightarrow wp(C, Q)$, we can obtain a finite counterexample to the implication.*

Proof. Checking the implication is equivalent to checking the unsatisfiability of the formula $P \wedge \neg wp(C, Q)$. Q is a $\forall^*\exists^*$-formula, so by Lemma 3.2, $wp(C, Q)$ is a $\forall^*\exists^*$-formula, and $\neg wp(C, Q)$ is an $\exists^*\forall^*$-formula. Thus, $P \wedge \neg wp(C, Q)$ is an $\exists^*\forall^*$-formula. Since all function symbols in RML are stratified, this formula is in EPR extended with stratified function symbols, and its satisfiability is decidable. Furthermore, this formula has the finite model property, and thus if it is satisfiable, we can obtain a finite model of it. □

4. An Interactive Methodology for Safety Verification

In this section, we describe our interactive approach for safety verification of RML programs. Recall that an RML program has the form

$$decls \; ; \; C_{\text{init}} \; ; \; \texttt{while} * \texttt{do} \; C_{\text{body}} \; ; \; C_{\text{final}}$$

We wish to verify the safety of the program, i.e. that it cannot reach an `abort` command. The core of this problem is to verify that all states reachable at the loop head will not lead to an `abort` [2], namely that they satisfy both $wp(C_{\text{final}}, true)$ and $wp(C_{\text{body}}, true)$.

The key idea is to combine automated analysis with user guidance in order to construct a universal inductive invariant that proves safety. The next subsection describes a preliminary stage for debugging the RML program, and the following subsections describe our interactive methodology of constructing universal inductive invariants.

4.1 Debugging via Symbolic Bounded Verification

Before starting the search for an inductive invariant, it makes sense to first search for bugs in the RML program. This can be done by unrolling the loop a bounded number of times. As we are most interested in states reachable at the loop head, we define an assertion φ to be k-*invariant* if it holds in all states reachable at the loop head after at most k loop iterations. Thus, φ is k-invariant iff:

$$\mathcal{A} \Rightarrow \bigwedge_{j=0}^{k} wp\left(C_{\text{init}} \; ; \; C_{\text{body}}^j, \varphi\right) \qquad (3)$$

Since the axioms $\mathcal{A}$ are $\exists^*\forall^*$-formulas, then by Theorem 3.3 checking the k-invariance of a $\forall^*\exists^*$-formula φ is decidable. Furthermore, if φ is not k-invariant, we can obtain a finite model of $\mathcal{A} \wedge \neg wp\left(C_{\text{init}} \; ; \; C_{\text{body}}^j, \varphi\right)$ for some $0 \leq j \leq k$. From this model, we can construct an execution trace that executes j loop iterations and reaches a state that violates φ. This trace can be graphically displayed to the user as a concrete counterexample to the invariance of φ. Note that while checking k-invariance bounds the number of loop iterations, it does not bound the size of the states. Thus, if a property is found to be k-invariant, it holds in *all* states reachable by k iterations. This is in contrast to finite-state bounded model checking techniques which bound the state space.

The first step when using Ivy is to model the system at hand as an RML program, and then debug the program by checking k-invariance of $wp(C_{\text{final}}, true)$ and $wp(C_{\text{body}}, true)$ (see Figure 3). If a counterexample is found, the user can examine the trace and modify the RML program

---

[2] Verifying that no execution of C_{init} leads to `abort` is simple as C_{init} executes only once, and amounts to checking if $\mathcal{A} \Rightarrow wp(C_{\text{init}}, true)$.

to either fix a bug in the code, or to fix a bug in the specification (the assertions). Once no more counterexample traces exist up to a bound that satisfies the user, the user moves to the second step of constructing a universal inductive invariant that proves the safety of the system for unbounded number of loop iterations.

4.2 Overview of the Interactive Search for Universal Inductive Invariants

Ivy assists the user in obtaining a universal inductive invariant. Recall that a formula I is an inductive invariant for the RML program if the conditions listed in Equation (2)) hold. Recall further that if I is a universally quantified formula, then by Theorem 3.3 these conditions are decidable, and if I does not satisfy them, we can obtain a finite state s that is a counterexample to one of the conditions. Such a state s is a *counterexample to induction (CTI)*.

Our methodology interactively constructs a universal inductive invariant represented as a conjunction of *conjectures*, i.e., $I = \bigwedge_{i=1}^{n} \varphi_i$. Each conjecture φ_i is a closed universal formula. In the sequel, we interchangeably refer to I as a set of conjectures and as the formula obtained from their conjunction.

Our methodology, presented in Figure 5, guides the user through an iterative search for a universal inductive invariant I by generalization from CTIs. We maintain the fact that all conjectures satisfy initiation ($\mathcal{A} \Rightarrow wp\left(C_{\texttt{init}}, \varphi_i\right)$), so each CTI we obtain is always a state that satisfies all conjectures, but leads either to an `abort` command by executing $C_{\texttt{body}}$ or $C_{\texttt{final}}$, or to a state that violates one of the conjectures by executing $C_{\texttt{body}}$.

Initialization   The search starts from a given set of conjectures as a candidate inductive invariant I. For example, I can be initialized to *true*. If $wp\left(C_{\texttt{final}}, true\right)$ and $wp\left(C_{\texttt{body}}, true\right)$ are universal, then I can initially include them (after checking that they satisfy initiation). If the search starts after a modification of the RML program (e.g. to fix a bug), then conjectures that were learned before can be reused. Additional initial conjectures can be computed by applying basic abstract interpretation techniques.

Iterations   Each iteration starts by (automatically) checking whether the current candidate I is an inductive invariant. If I is not yet an inductive invariant, a CTI is presented to the user, which is a state s that either leads to an `abort` via $C_{\texttt{final}}$ (safety violation), or leads to a state that violates I via $C_{\texttt{body}}$ (consecution violation). The user can choose to strengthen I by conjoining it with an additional conjecture that excludes the CTI from I. To this end, we provide automatic mechanisms to assist in *generalizing* from the CTI to obtain a conjecture that excludes it. In case of a consecution violation, the user may also choose to weaken I by eliminating one (or more) of the conjectures. The user can also choose to modify the RML program in case the CTI

indicates a bug. This process continues until an inductive invariant is found.

For I to be an inductive invariant, all the conjectures φ_i need to be invariants (i.e., hold in all states reachable at the loop head). This might not hold in the intermediate steps of the search, but it guides strengthening and weakening: strengthening aims at only adding conjectures that are invariants, and weakening aims at identifying and removing conjectures that are not invariants. While strengthening and weakening are ultimately performed by the user, we provide several automatic mechanisms to assist the user in this process.

Obtaining a minimal CTI   When I is not inductive, it is desirable to present the user with a CTI that is easy to understand, and not cluttered with many unnecessary features. This also tends to lead to better generalizations from the CTI. To this end, we automatically search for a "minimal" CTI, where the minimization parameters are defined by the user (see Section 4.3).

Interactive generalization   To eliminate the CTI s, the user needs to either strengthen I to exclude s from I, or, in case of a consecution violation, to weaken I to include a state reachable from s via $C_{\texttt{body}}$. Intuitively, if s is not reachable, then I should be strengthened to exclude it. If s is reachable, then I should be weakened. Clearly, checking whether s is reachable is infeasible. Instead, we provide the user with a generalization assistance for coming up with a new conjecture to strengthen I. The goal is to come up with a conjecture that is satisfied by all the reachable states. During the attempt to compute a generalization, the user might also realize that an existing conjecture is in fact not an invariant (i.e., it is not satisfied by all reachable states), and hence weakening is in order. In addition, the user might also find a modeling bug which means the RML program should be fixed.

Generalizations are explained in Section 4.4, and the interactive generalization process is explained in Section 4.5. Here again, the user defines various parameters for generalization, and Ivy automatically finds a candidate that meets the criteria. The user can further change the suggested generalization and can use additional automated checks to decide whether to keep it.

Remark 4.1. If all the conjectures added by the user exclude only unreachable states (i.e., all are invariants), then weakening is never needed.   As such, most of the automated assistance we provide focuses on helping the user obtaining "good" conjectures for strengthening—conjectures that do not exclude reachable states. Weakening will typically be used when some conjecture turns out to be "wrong" in the sense that it does exclude reachable states.

4.3 Obtaining Minimal CTIs

We refine the search for CTIs by trying to find a minimal CTI according to user adjustable measures. As a general rule, smaller CTIs are desirable since they are both easier to under-

stand, which is important for interactive generalization, and more likely to result in more general (stronger) conjectures. The basic notion of a small CTI refers to the number of elements in its domain. However, other quantitative measures are of interest as well. For example, it is helpful to minimize the number of elements (or tuples) in a relation, e.g. if the relation appears as a guard for protocol actions (such as the *pnd* relation in the leader election protocol of Figure 1). Thus, we define a set of useful minimization measures, and let the user select which ones to minimize, and in which order.

Minimization measures   The considered measures are:
- Size of sort S: $|D_S|$ where D_S is the domain of sort S.
- Number of positive tuples of r: $|\{\overline{e} \mid \mathcal{I}(r)(\overline{e}) = 1\}|$.
- Number of negative tuples of r: $|\{\overline{e} \mid \mathcal{I}(r)(\overline{e}) = 0\}|$.

Each measure m induces an order $\leq_m$ on structures, and each tuple $(m_1, \ldots, m_t)$ of measures induces a "smaller than" relation on structures which is defined by the lexicographic order constructed from the orders $\leq_{m_i}$.

Minimization procedure   Given the tuple of measures to minimize, provided by the user, Algorithm 1 automatically finds a CTI that is minimal with respect to this lexicographic order. The idea is to conjoin ψ_{cti} (which encodes violation of inductiveness) with a formula ψ_{min} that is computed incrementally and enforces minimality. For a measure m, $\varphi_m(n)$ is an $\exists^* \forall^*$ clause stating that the value of m is no more than n. Such constraints are added to ψ_{min} for every m, by their order in the tuple, where n is chosen to be the minimal number for which the constraint is satisfiable (with the previous constraints). Finally, a CTI that obeys all the additional constraints is computed and returned.

For example, consider a k-ary relation r. We encode the property that the number of positive tuples of r is at most n as follows: $\exists \overline{x_1}, \ldots \overline{x_n}. \forall \overline{y}. \left(r(\overline{y}) \rightarrow \bigvee_{i=1}^{n} \overline{y} = \overline{x_i} \right)$, where $\overline{x_1}, \ldots, \overline{x_n}, \overline{y}$ denote k-tuples of logical variables.

Algorithm 1: Obtaining a Minimal CTI

---

1 **if** ψ_{cti} *is unsatisfiable* **then return** None;
2 $\psi_{min} := true$;
3 **for** m *in* $(m_1, \ldots, m_t)$ **do**
4   **for** n *in* $0, 1, 2, \ldots$ **do**
5   **if** $\psi_{cti} \wedge \psi_{min} \wedge \varphi_m(n)$ *is satisfiable* **then**
6              $\psi_{min} := \psi_{min} \wedge \varphi_m(n)$;
7   **break**
8 **return** s such that $s \models \psi_{cti} \wedge \psi_{min}$

4.4 Formalizing Generalizations as Partial Structures

In this subsection we present the notion of a *generalization* and the notion of a *conjecture* associated with a generalization. These notions are key ingredients in the interactive generalization step described in the next subsection.

Recall that a CTI is a structure. Generalizations of a CTI are given by partial structures, where relation symbols and function symbols are interpreted as *partial* functions. Formally:

Definition 2 (Partial Structures). Given a vocabulary Σ and a domain D, a *partial interpretation function* $\mathcal{I}$ of Σ over D associates every k-ary relation symbol $r \in \Sigma$ with a *partial* function $\mathcal{I}(r) : D^k \rightharpoonup \{0, 1\}$, and associates every k-ary function symbol $f \in \Sigma$ with a *partial* function $\mathcal{I}(f) : D^{k+1} \rightharpoonup \{0, 1\}$ such that for any $x_1, \ldots, x_k \in D$, $\mathcal{I}(f)(x_1, \ldots, x_k, y) = 1$ for *at most* one element $y \in D$. $\mathcal{I}$ must also obey the sort restrictions.

A *partial structure* over Σ is a pair $(D, \mathcal{I})$, where $\mathcal{I}$ is a partial interpretation function of Σ over domain D.

Note that a structure is a special case of a partial structure. Intuitively, generalization takes place when a CTI is believed to be unreachable, and a partial structure generalizes a CTI (structure) by turning some values ("facts") to be undefined or unspecified. For example, in a partial structure, $\mathcal{I}(r)(\overline{e})$ might remain undefined for some tuple $\overline{e}$. This is useful if the user believes that the structure is still unreachable, regardless of the value of $\mathcal{I}(r)(\overline{e})$. In Figures 7 to 9, (a1) and (a2) always represent total structures, while (b) and (c) represent partial structures.

A natural *generalization partial order* can be defined over partial structures:

Definition 3 (Generalization Partial Order). Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be two partial interpretation functions of Σ over D_1 and D_2 respectively, such that $D_2 \subseteq D_1$. We say that $\mathcal{I}_2 \sqsubseteq \mathcal{I}_1$ if for every k-ary relation or function symbol $a \in \Sigma$, If $\overline{e} \in dom(\mathcal{I}_2(a))$, then $\overline{e} \in dom(\mathcal{I}_1(a))$ as well, and $\mathcal{I}_2(a)(\overline{e}) = \mathcal{I}_1(a)(\overline{e})$.

For partial structures $s_1 = (D_1, \mathcal{I}_1)$ and $s_2 = (D_2, s_2)$ of Σ, we say that $s_2 \sqsubseteq s_1$ if $D_2 \subseteq D_1$ and $\mathcal{I}_2 \sqsubseteq \mathcal{I}_1$.

The generalization partial order extends the substructure relation of (total) structures. Intuitively, $s_2 \sqsubseteq s_1$ if the interpretation provided by s_1 is at least as "complete" (defined) as the interpretation provided by s_2, and the two agree on elements (or tuples) for which s_2 is defined. Thus, $s_2 \sqsubseteq s_1$ when s_2 represents more states than s_1, and we say that s_2 is a generalization of s_1.

From partial structures to conjectures   Intuitively, every partial structure s represents an infinite set of structures that are more specific than s (they interpret more facts), and the conjecture that a partial structure induces excludes all these structures (states). Formally, a partial structure induces a universally quantified conjecture that is obtained as the negation of the *diagram* of the partial structure, where the classic definition of a diagram of a structure is extended to partial structures.

Definition 4 (Diagram). Let $s = (D, \mathcal{I})$ be a finite partial structure of Σ and let $D' = \{e_1, \ldots, e_{|D'|}\} \subseteq D$ denote the set of elements e_i for which there exists (at least one) relation or function symbol $a \in \Sigma$ such that e_i appears in the domain of definition of $\mathcal{I}(a)$. The *diagram* of s, denoted by $Diag(s)$, is the following formula over Σ:

$$\exists x_1 \ldots x_{|D'|}. \mathtt{distinct}(x_1 \ldots x_{|D'|}) \wedge \psi$$

where ψ is the conjunction of:

- $r(x_{i_1}, \ldots, x_{i_k})$ for every k-ary relation r in Σ and every $i_1, \ldots, i_k$ s.t. $\mathcal{I}(r)(e_{i_1}, \ldots, e_{i_k}) = 1$, and
- $\neg r(x_{i_1}, \ldots, x_{i_k})$ for every k-ary relation r in Σ and every $i_1, \ldots, i_k$ s.t. $\mathcal{I}(r)(e_{i_1}, \ldots, e_{i_k}) = 0$, and
- $f(x_{i_1}, \ldots, x_{i_k}) = x_j$ for every k-ary function f in Σ and every $i_1, \ldots, i_k$ and j s.t. $\mathcal{I}(f)(e_{i_1}, \ldots, e_{i_k}, e_j) = 1$, and
- $f(x_{i_1}, \ldots, x_{i_k}) \neq x_j$ for every k-ary function f in Σ and every $i_1, \ldots, i_k$ and j s.t. $\mathcal{I}(f)(e_{i_1}, \ldots, e_{i_k}, e_j) = 0$.

Intuitively, $Diag(s)$ is obtained by treating individuals in D as existentially quantified variables and explicitly encoding all the facts that are defined in s.

The negation of the diagram of s constitutes a *conjecture* that is falsified by all structures that are more specific than s. This includes all structures that contain s as a substructure.

Definition 5 (Conjecture)**.** Let s be a partial structure. The *conjecture associated with s*, denoted $\varphi(s)$, is the universal formula equivalent to $\neg Diag(s)$.

Lemma 4.2. *Let s be a partial structure and let s' be a (total) structure such that $s \sqsubseteq s'$. Then $s' \not\models \varphi(s)$.*

Note that if $s_2 \sqsubseteq s_1$, then $\varphi(s_2) \Rightarrow \varphi(s_1)$ i.e., a larger generalization results in a stronger conjecture.

This connection between partial structures and conjectures is at the root of our graphical interaction technique. We present the user with partial structures, and the user can control which facts to make undefined, thus changing the partial structure. The semantics of the partial structure is given by the conjecture associated with it. The conjectures C_1, C_2, and C_3 of Figure 6 are the conjectures associated with the partial structures depicted in Figure 7 (c), Figure 8 (b), and Figure 9 (c) respectively.

4.5 Interactive Generalization

Generalization of a CTI s in Ivy consists of the following conceptual phases that are controlled by the user:

Coarse-grained manual generalization The user graphically selects an upper bound for generalization $s_u \sqsubseteq s$, with the intent to obtain a $\sqsubseteq$-smallest generalization s' of s_u. Intuitively, the upper bound s_u defines which elements of the domain may participate in the generalization and which tuples of which relations may stay interpreted in the generalization. For example, if a user believes that the CTI remains unreachable even when some $\mathcal{I}(r)(\bar{e})$ is undefined, they can use this intuition to define the upper bound.

In Ivy the user defines s_u by graphically marking the elements of the domain that will remain in the domain of the partial structure. In addition, for every relation or function symbol a, the user can choose to turn all positive instances of $\mathcal{I}(a)$ (i.e., all tuples $\bar{e}$ such that $\mathcal{I}(a)(\bar{e}) = 1$) to undefined, or they can choose to turn all negative instances of $\mathcal{I}(a)$ to undefined. The user makes such choices by selecting appropriate checkboxes for every symbol.

In Figures 7 to 9, (b) depicts the upper bound s_u selected by the user according to the user's intuition.

Fine-grained automatic generalization via k-invariance Ivy searches for a $\sqsubseteq$-smallest generalization s' that generalizes s_u such that $\varphi(s')$ is k-invariant (i.e., s' is unreachable in k steps), where k is provided by the user.

This process begins by checking if $\varphi(s_u)$ is k-invariant using Equation (3). If verification fails, the user is presented with a trace that explains the violation. Based on this trace, the user can either redefine s_u to be less general, or they may decide to modify the RML program if a bug is revealed.

If $\varphi(s_u)$ is k-invariant, it means that the negation of Equation (3) for $\varphi(s_u)$ is unsatisfiable. In this case, Ivy computes the minimal UNSAT core out of the literals of $\varphi(s_u)$ and uses it to define a most general ($\sqsubseteq$-smallest) s_m such that $\varphi(s_m)$ is still k-invariant. The partial structure s_m obtained by the UNSAT core is displayed to the user as a candidate generalization (the user can also see the corresponding conjecture).

The partial structures, s_m, obtained in this stage are depicted in Figures 7 and 9 (c). For the CTI of Figure 8, s_u, depicted in (b), is already minimal, and the UNSAT core is not able to remove any literals (so in this case $s_u = s_m$).

User investigates the suggested generalization After the automatic generalization found a stronger conjecture $\varphi(s_m)$ that is still k-invariant, the user must decide whether to add this conjecture to the candidate inductive invariant I. In order to make this decision, the user can check additional properties of the generalization (and the conjecture associated with it). For example, the user may check if it is k'-invariant for some $k' > k$. The user can also examine both the graphical visualization of s_m and a textual representation of $\varphi(s_m)$ and judge it according to their intuition about the system.

If the obtained conjecture does not seem correct, the user can choose to increase k and try again. If a conjecture that appears bogus remains k-invariant even for large k, it may indicate a bug in the RML program that causes the behaviors of the program to be too restricted (e.g. an axiom or an assume that are too strong). The user may also choose to manually fine-tune the conjecture by re-introducing interpretations that became undefined. The user can also choose to change the generalization upper bound s_u or even ask Ivy for a new CTI, and start over. Eventually, the user must decide on a conjecture to add to I for the process to make progress.

5. Initial Experience

In this section we provide an empirical evaluation of the approach presented above. Ivy is implemented in Python and uses Z3 [4] for satisfiability testing. Ivy supports both the procedure for symbolic bounded verification described in Section 4.1 and the procedures for interactive construction of inductive invariants described in Sections 4.2 to 4.5. Ivy provides a graphical user interface implemented using JavaScript in an IPython [25] notebook.

5.1 Protocols

Lock server We consider a simple lock server example taken from Verdi [28, Fig. 3]. The system contains an unbounded number of clients and a single server. Each client has a flag that denotes whether it thinks it holds the lock or not. The server maintains a list of clients that requested the lock, with the intent of always granting the lock to the client at the head of the list. A client can send a lock request to the server. When this request is received the server adds the client to the end of the list. If the list was previously empty, the server will also immediately send back a grant message to the client. A client that holds the lock can send an unlock message that, when received, will cause the server to remove the client from the waiting list, and send a grant message to the new head of the list. In this protocol, messages cannot be duplicated by the network, but they can be reordered. Consequently, the same client can appear multiple time in the server's waiting list. The safety property to verify is that no two clients can simultaneously think they hold the lock.

Distributed lock protocol Next, we consider a simple distributed lock protocol taken from [8, 12] that allows an unbounded set of nodes to transfer a lock between each other without a central server. Each node maintains an integer denoting its current epoch and a flag that denotes if it currently holds the lock. A node at epoch e that holds the lock, can transfer the lock to another node by sending a transfer message with epoch $e + 1$. A node at epoch e that receives a transfer message with epoch e' ignores it if $e' \leq e$, and otherwise it moves to epoch e', takes the lock, and sends a locked message with epoch e'. In this protocol, messages can be duplicated and reordered by the network. The safety property to verify is that all locked messages within the same epoch come from a single node.

Learning switch Learning switches are a basic component in networking. A learning switch maintains a table, used to route incoming packets. On receiving a packet, the learning switch adds a table entry indicating that the source address can be reached by forwarding out the incoming port. It then checks to see if the destination address has an entry in the table, and if so forwards the packet using this entry. If no entry exists it floods the packet out all of its ports with the exception of the incoming port. For this protocol we check whether the routing tables for all switches could contain a forwarding loops. We consider a model with an unbounded number of switches and an unbounded forwarding table. routing table of each switch contains an unbounded number of entries.

We model the network using a binary relation *link* describing the topology of the network and a 4-arity relation *pending* of the set of all packets pending in the network; $pending(s, d, sw_1, sw_2)$ implies a packet with source s and destination d is pending to be received along the sw_1–sw_2 link. We store the routing tables using relations *learned*, and *route*. For verification we use *route** a relation which mod-

els the reflexive transitive closure of *route*. The modeling maintains *route** using the standard technique for updating transitive closure (e.g. [14]). The safety property is specified by an assertion that whenever a switch learns a new route, it does not introduce a cycle in the forwarding graph.

Database chain consistency Transaction processing is a common task performed by database engines. These engines ensure that (a) all operations (reads or writes) within a transaction appear to have been executed at a single point in time (*atomicity*), (b) a total order can be established between the committed transactions for a database (*serializability*), and (c) no transaction can read partial results from another transaction (*isolation*). Recent work (e.g., [30]) has provided a chain based mechanism to provide these guarantees in multi-node databases. In this model the database is sharded, i.e., each row lives on a single node, and we wish to allow transactions to operate across rows in different nodes.

Chain based mechanisms work by splitting each transaction into a sequence of *subtransactions*, where each subtransaction only accesses rows on a single node. These subtransactions are executed sequentially, and traditional techniques are used to ensure that subtransaction execution does not violate safety conditions. Once a subtransaction has successfully executed, we say it has precommitted, i.e., the transaction cannot be aborted due to command in the subtransaction. Once all subtransactions in a transaction have precommitted, the transaction itself commits, if any subtransaction aborts the entire transaction is aborted. We used Ivy to show that one such chain transaction mechanism provides all of the safety guarantees provided by traditional databases.

The transaction protocol was modeled in RML using a sort for *transaction*, *node*, *key* (row) and *subtransaction*. Commit times are implicitly modeled by transactions (since each transaction has a unique commit time), and unary relations are used to indicate that a transaction has committed or aborted. We modeled the sequence of subtransactions in a transaction using the binary relation *opOrder* and tracked a transactions dependencies using the binary relation *writeTx* (indicating a transaction t wrote to a row) and a ternary relation *dependsTx* (indicating transaction t read a given row, and observed writes from transaction t'). To this model we added assertions ensuring that (a) a transaction reading row r reads the last committed value for that row, and (b) uncommitted values are not read. For our protocol this is sufficient to ensure atomicity.

Chord ring maintenance Chord is a peer-to-peer protocol implementing a distributed hash table. In [29], Zave presented a model of the part of the protocol that implements a self-stabilizing ring. This was proved correct for the case of up to 8 participants, but the parameterized case was left open. We modeled Chord in Ivy and attempted to prove the primary safety property, which is that the ring remains connected under certain assumptions about failures. Our method was similar to Houdini [6] in that we described a class of formulas using a template, and used abstract interpretation

Protocol	S	RF	C	I	G
Leader election in ring	2	5	3	12	3
Lock server	5	11	3	21	8
Distributed lock protocol	2	5	3	26	12
Learning switch	2	5	11	18	3
Database chain replication	4	13	11	35	7
Chord ring maintenance	1	13	35	46	4

Figure 14. Protocols verified interactively with Ivy. **S** is the number of sorts in the model. **RF** is the number of relations and function symbols in the model. **C** is the size of the initial set of conjectures, measured by the total number of literals that appear in the formulas. **I** is the size of the final inductive invariant (also measured by total number of literals). **G** is the number of CTI's and generalizations that took place in the interactive search for the inductive invariant.

to construct the strongest inductive invariant in this class. This was insufficient to prove safety, however. We took the abstract state at the point the safety failure occurred as our attempted inductive invariant, and used Ivy's interaction methods to diagnose the proof failure and correct the invariant. An interesting aspect of this proof is that, while Zave's proof uses the transitive closure operator in the invariant (and thus is outside any known decidable logic) we were able to interactively infer a suitable universally quantified inductive invariant.

5.2 Results & Discussion

Next, we evaluate Ivy's effectiveness. We begin, in Figure 14 by quantifying model size, size of the inductive invariant discovered and the number of CTIs generated when modeling the protocols described above. As can be seen, across a range of protocols, modeled with varying numbers of sorts and relation and function symbols, Ivy allowed us to discover inductive invariants in a modest number of interactive steps (as indicated by the number of CTIs generated in column **G**). However, Ivy is highly interactive and does not cleanly lend itself to performance evaluation (since the human factor is often the primary bottleneck). We therefore present here some observations from our experience using Ivy as an evaluation into its utility.

Modeling Protocols in Ivy Models in Ivy are written in an extended version of RML. Since, RML and Ivy are restricted to accepting code that can be translated into EPR formulas, they force some approximation on the model. For example, in the database commit protocol, expressing a constraint requiring that every subtransaction reads or writes at least one row is impossible in EPR, and we had to overapproximate to allow empty subtransactions.

Bounded Verification Writing out models is notoriously error prone. We found Ivy's bounded verification stage to be

invaluable while debugging models, even when we bounded ourselves to a relatively small number of steps (typically 3 – 9). Ivy displays counterexamples found through bounded verification using the same graphical interface as is used during inductive invariant search. We found this graphical representation of the counterexample made it easier to understand modeling bugs and greatly sped up the process of debugging a model.

Finding Inductive Invariants In our experience, using a graphical representation to select an upper bound for generalization was simple and the ability to visually see a concrete counterexample allowed us to choose a much smaller partial structure. In many cases we found that once a partial structure had been selected, automatic generalization quickly found the final conjecture accepted by the user.

Additionally, in some cases the conjectures suggested by Ivy were too strong, and indicated that our modeling excluded valid system behaviors. For example, when modeling the database example we initially used `assume`'s that were too strong. We detected this when we saw Ivy reporting that a conjecture that seemed bogus is true even for a high number of transitions. After fixing the model, we could reuse work previously done, as many conjectures remained invariant after the fix.

Comparing safety verification in Ivy to Coq and Dafny The lock server protocol is taken from [28], and thus allows some comparison of the safety verification process in Ivy to Coq and the Verdi framework. The size and complexity of the protocol description in Ivy is similar to Verdi, and both comprise of approximately 50 lines of code. When verifying safety with Verdi, the user is required to manually think of the inductive invariant, and then prove its inductiveness using Coq. For this protocol, [28] reports a proof that is approximately 500 lines long. With Ivy, the inductive invariant was found after 8 iterations of user guided generalizations, which took us less then an hour. Note that with Ivy, there is no need to manually prove that the invariant is inductive, as this stage is fully automated.

The distributed lock protocol is taken from [8], which allows a comparison with Dafny and the IronFleet framework. This protocol took us a few hours to verify with Ivy. Verifying this protocol with Dafny took the authors of [8] a few days, when a major part of the effort was manually coming up with the inductive invariant [24]. Thus, for this protocol, the help Ivy provides in finding the inductive invariant significantly reduces the verification effort.

In both cases we are comparing the substantial part of the proof, which is finding and proving the inductive invariant. There are some differences in the encoding of this problem, however. For example, we use a totally ordered set where the Coq version of the lock server example uses a list. From the Coq version, executable code can be extracted, whereas we cannot currently do this from the Ivy version.

Overall Thoughts We believe Ivy makes it easier for users to find inductive invariants, and provides a guided experience through this process. This is in contrast to the existing model for finding inductive invariants, where users must come up with the inductive invariant by manual reasoning.

6. Related Work

The idea of using decidable logics for program verification is quite old. For example, Klarlund *et al.* used monadic second order (MSO) logic for verification in the Mona system [9]. This approach has been generalized in the STRAND logic [22]. Similar logics could be used in the methodology we propose. However, EPR has the advantage that it does not restrict models to have a particular structure (for example a tree or list structure). Moreover as we have seen there are simple and effective heuristics for generalizing a counterexample model to an EPR formula. Finally, the complexity of EPR is relatively low (exponential compared to non-elementary) and it is implemented in efficient provers such as Z3.

Various techniques have been proposed for solving the parameterized model checking problem (PMCP). Some achieve decidability by restricting the process model to specialized classes that have cutoff results [7] or can be reduced to well-structured transition systems (such as Petri nets) [1]. Such approaches have the advantages of being fully automated when they apply. However, they have high complexity and do not fail visibly. This and the restricted model classes make it difficult to apply these methods in practice.

There are also various approach to the PMCP based on abstract interpretation. A good example is the Invisible Invariants approach [27]. This approach attempts to produce an inductive invariant by generalizing from an invariant of a finite-state system. However, like other abstract interpretation methods, it has the disadvantage of not failing visibly.

The kind of generalization heuristic we use here is also used in various model checking techniques, such IC3 [2]. A generalization of this approach called UPDR can automatically synthesize universal invariants [17]. The method is fragile, however, and we were not successful in applying it to the examples verified here. Our goal in this work is to make this kind of technique interactive, so that user intuition can be applied to the problem.

There are also many approaches based on undecidable logics that provide varying levels of automation. Some examples are proof assistants such as Coq that are powerful enough to encode all of mathematics but provide little automation, and tools such as Dafny [19] that provide incomplete verification condition checking. The latter class of tools provide greater automation but do not fail visibly. Because of incompleteness they can produce incorrect counterexample models, and in the case of a true counterexample to induction they provide little feedback as to the root cause of the proof failure.

7. Conclusion

We have presented Ivy — a tool for interactively verifying safety of infinite-state systems. Ivy is based on two key insights. First, the modeling language should be designed to permit effective abstract specification and yet allow decidable invariant checking using satisfiability solvers. Second, invariant generation is fundamentally based on generalization, which should be a collaborative process between the human user and automated mechanisms. Ivy has many other features not discussed here, including abstract interpretation, support for modular refinement proofs, test generation and extraction of executable implementations.

We hope that Ivy will become a useful tool for system builders and designers. We are encouraged by our initial experience in protocols like chain replication, where the designer did not initially know what safety properties should be satisfied or how to formally model the protocol. We found the graphical interface to be extremely valuable as a proof interaction tool. In the case of the Chord protocol, it also led us to a simplified proof in the unbounded setting.

Although we focused on EPR in this paper, the methodology of generating inductive invariants by interactive generalization from CTIs is more general. It depends only on the following three properties of the modeling language and the class of conjectures: (i) decidability of bounded verification — checking if a conjecture is true after k transitions, (ii) decidability of checking inductiveness of a candidate conjecture set, and (iii) the ability to graphically represent conjectures in a way that is intuitive for system developers, and allows them to graphically define generalizations. RML together with universal conjectures satisfy these properties. We hope that our methodology will also be applied in other verification settings where these properties can be satisfied.

Acknowledgments

References

[1] P. A. Abdulla, K. Cerans, B. Jonsson, and Y. Tsay. Algorithmic analysis of programs with well quasi-ordered domains. *Inf. Comput.*, 160(1-2):109–127, 2000.

[2] A. R. Bradley. Sat-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI*, pages 70–87, 2011.

[3] E. Chang and R. Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.

[4] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[6] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe*, pages 500–517, 2001.

[7] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992.

[8] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP*, pages 1–17, 2015.

[9] J. G. Henriksen, J. L. Jensen, M. E. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS*, pages 89–110, 1995.

[10] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[11] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant a tutorial. *Rapport Technique*, 178, 1997.

[12] IronFleet Project. Distributed lock service protocol source code. `https://github.com/Microsoft/Ironclad/blob/40b281f9f9fa7cfca5a00a7085cb302e6b1a9aa6/ironfleet/src/Dafny/Distributed/Protocol/Lock/Node.i.dfy`. Accessed: 2016-03-20.

[13] S. Itzhaky, A. Banerjee, N. Immerman, O. Lahav, A. Nanevski, and M. Sagiv. Modular reasoning about heap paths via effectively propositional formulas. In *the 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 385–396, 2014.

[14] S. Itzhaky, A. Banerjee, N. Immerman, A. Nanevski, and M. Sagiv. Effectively-propositional reasoning about reachability in linked data structures. In *Computer Aided Verification - 25th International Conference, CAV*, pages 756–772, 2013.

[15] Ivy PLDI'16 web page. `https://www.cs.tau.ac.il/~odedp/ivy/`.

[16] D. Jackson. *Software Abstractions: Resources and Additional Materials*. MIT Press, 2011.

[17] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. In *Computer Aided Verification - 27th International Conference, CAV*, pages 583–602, 2015.

[18] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010.

[19] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, pages 348–370, 2010.

[20] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[21] H. R. Lewis. Complexity results for classes of quantificational formulas. *Journal of Computer and System Sciences*, 21(3):317 – 353, 1980.

[22] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, pages 43–59, 2011.

[23] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[24] B. Parno. private communication, 2016.

[25] F. Pérez and B. E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[26] R. Piskac, L. M. de Moura, and N. Bjørner. Deciding effectively propositional logic using DPLL and substitution sets. *J. Autom. Reasoning*, 44(4):401–424, 2010.

[27] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS*, pages 82–97, 2001.

[28] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[29] P. Zave. How to make chord correct (using a stable base). *CoRR*, abs/1502.06461, 2015.

[30] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction chains: achieving serializability with low latency in geo-distributed storage systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 276–291. ACM, 2013.