Ivy: Interactive Verification of Parameterized Systems via Effectively Propositional Reasoning

Abstract

The design and implementation of parametric systems can be very tricky even for experienced researchers. We describe an interactive system — Ivy — for interactively verifying parameterized systems. Ivy is based on the following principles: (i) Ivy first attempts to locate counterexamples by bounding the number of protocol actions and symbolically searching for (unbounded) bad inputs. (ii) Invariants in Ivy are expressed as universal formulas in relational first-order logic. Their inductiveness check reduces to unsatisfiability checks in Effectively Propositional Logic (EPR). This guarantees that the tool can always decide whether an invariant is inductive or not. Furthermore, our use of universal formulas guarantees that counterexamples to induction can be presented graphically, allowing inspection by humans. (iii) This allows users to guide the verification process by suggesting candidate local invariants, whose conjunction comprises a global inductive invariant. All user interactions are performed using graphical models, easing the user's task. We describe our initial experience with some protocols.

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 at verification of real-world systems use relatively little proof automation [7, 13, 15]. 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. 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 for this process to 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, however. To make such failures visible, we propose a technique of graphically visualizing counterexamples to induction that makes the generalization process visible and allows the user to guide it.

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 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 [7]). To eliminate this sort of invisible failure, we will consider implications of restricting the specification language in such a way that all verification conditions fall into a decidable fragment of the logic: the Bernays-Shoenfinkel-Ramsey fragment, also known as EPR [19].

We test these ideas by implementing them in a verification tool and applying it interactively to a variety of infinite-state or parameterized systems. Although we are unable to prove correctness of these systems in a fully automated way, we find that automated generalization heuristics are still useful when applied with human guidance. Moreover, we find that the restriction to EPR is not an impediment to constructing proofs, although it may require the user to define auxiliary predicates in order to express the required invariant. The compensation for this is that in no case were we unable to check a verification condition using the Z3 theorem prover.

Main Results The contributions of this paper are:

- A modeling language for distributed parameterized protocols called RML. RML is a modeling language inspired by Alloy [11]. It represents program states as sets of first order relations. Updates to relations are restricted to be quantifier free. Non-deterministic statements in the style of Boogie and Dafny are supported. RML is designed to guarantee that the transition relation for every loop free program fragment is EPR expressible.
- The usage of EPR to enable bounded verification of RML programs which only bounds the number of transitions but not the states.
- A new methodology for unbounded safety verification via an interactive search for universal inductive invariants, and its realization for RML programs. Our methodolody combines user guidance with automated reasoning in a decidable logic.
- An initial evaluation of the methodology on some interesting distributed protocols.

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 [17] and Coq [9] 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 [12]. Ivy aims to balance between the predictability and visibility of proof assistants, and the automation of decision procedures and invariant inference heuristics.

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. Specifically, Ivy only uses satisfiability queries of the form $\exists^*.\forall^*.\varphi_{\mathtt{QF}}$ where $\varphi_{\mathtt{QF}}$ is a quantifier-free formula over uninterpreted relations. These are called **E**ffectively **Pr**opotional Formulas (EPR) [19] and comprise one of the simplest decidable fragments of first order logic. The use of EPR formulas has many benefits outlined in Section 2.4. The restriction to EPR is achieved by excluding from RML many convenient constructs such as arrays, functions and arithmetic operations. However, as we show, it is possible to reason in a sound and complete way about many protocol features including partial and total orders and deterministic transitive closure [10].

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. A node only forwards messages with higher ID than its own ID. When a node receives a message with its own ID, it declares itself as a leader.

The RML program uses variables and relations to model the state of the protocol which evolves over time. The variable *head* holds the head node of the ring and *tail* holds the tail of the ring. The variable *leader_id* is a ghost variable which records the identity of the leader. A binary relation *le* encodes a total order on IDs. The topology is represented by a total order *reach* on nodes in the ring where x *reach* y holds if there is a path from x to y which does not require traversing the edge from *tail* to *head*. *le* and *reach* are modeled as uninterpreted relations which are axiomatized in a sound and complete way to be total orders. The unary relation *leader* holds for nodes which are identified as leaders. The binary relation *pending* between nodes and IDs denotes pending messages. The safety property of the protocol, stated via an `assert` command, is that there is at most one leader. We consider two more user-provided conjectures, stating that the leader has the highest ID of all nodes, and that *leader_id* holds the ID of the leader.

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 is debugging the protocol symbolically. Bounded model checking tools such as Alloy [11] 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 from an initial state can lead to an error. An error can either be a violation of the safety property or a violation of a conjecture. This phase is depicted in Figure 2. In our experience this phase is very effective for bug finding since often the protocol and/or the desired properties are wrong. In the leader election example, we ran Ivy with 10 transitions to debug the model and the safety property. Notice again that Ivy does not restrict the size of the ring. In our experience, protocols can be verified for about 10 transitions in a few minutes which is tolerable.

```
sort Node; sort Id;
var head:Node; var tail:Node; var leader_id:Id;
rel le:Id,Id; rel reach:Node,Node; rel id:Node,Id;
rel pending:Id,Node; rel leader:Node;
assume !leader(N) && !pending(I,N) && r_axioms;
while * do {
  assert !(leader(N1) && leader(N2) && N1 != N2);
  {
    n1 := *; n2 := *; i := *;
    assume next_in_ring(n1, n2) && id(n1, i);
    pending.insert(i, n2);
  } | {
    m := *; n1:= *; n2 := *; i1 := *;
    assume pending(m, n1) && id(n1, i1);
    skip | pending.remove(m, n1);
    if i1 = m then {
      leader.insert(n1); leader_id := i1;
    } else {
      if le(i1, m) then {
        assume next_in_ring(n1, n2);
        pending.insert(m, n2)
      } else skip } } }
```

Figure 1. An RML program modeling the leader election protocol. Capital letters denote universally quantified logical variables. `next_in_ring(n1, n2)` denotes a universal formula expressing the fact `n2` is the immediate neighbor of `n1`. `r_axioms` denotes a universal formula expressing that `le` and `reach` are total orders, and that `head` is minimal and `tail` is maximal with respect to `reach`.

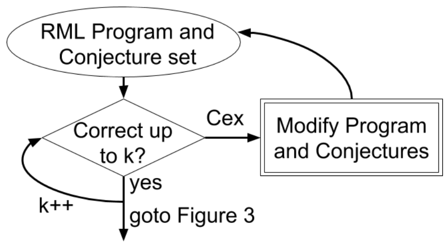
RML Program and Conjecture set
Correct up to k?
Cex
Modify Program and Conjectures
yes
k++
goto Figure 3

Figure 2. Flowchart of bounded verification.

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 Inductive Invariants

The second phase of the verification is to find an inductive invariant that proves that the protocol 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 for the program satisfy I (**initiation**). (ii) Every state satisfying I also satisfies the desired safety properties (**safety**). (iii) I is closed under the transition system, i.e., executing a transition from any arbitrary program state satisfying I results in a new program state which also satisfies I (**consecution**).

If the user has a universal inductive invariant in mind, Ivy can automatically check if it is indeed an inductive invariant. This check is guaranteed to terminate with either a proof showing that the invariant is inductive or a counterexample which can be depicted graphically and presented to the user. We refer to such a counterexample as a **C**ounterexample **t**o **I**nduction (CTI). A CTI does not necessarily imply that the safety property is violated — only that I does not satisfy the above conditions. Coming up with inductive invariants in distributed protocols is very difficult. Therefore, Ivy supports an interactive procedure for gradually obtaining an inductive invariant or deciding that the protocol or the safety property need to be revised.

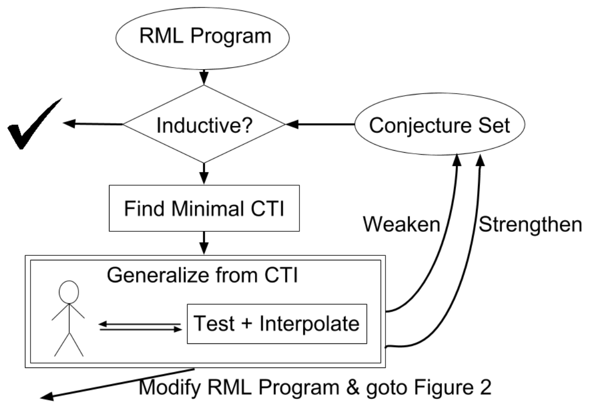

Figure 3. Flowchart of the interactive search for an inductive invariant.

The search for inductive invariants starts with a set of conjectures, and advances based on CTIs according to the procedure described in Figure 3. 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 2. 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.

Obtaining helpful CTI's Since we rely on the user to guide the generalization, it is critical to display a CTI which is easy to understand and that is 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 are minimizing the number of elements, and minimizing certain relations. Figure 4(left) shows the CTI obtained by applying `Find Minimal CTI` in the leader election example in the first iteration.

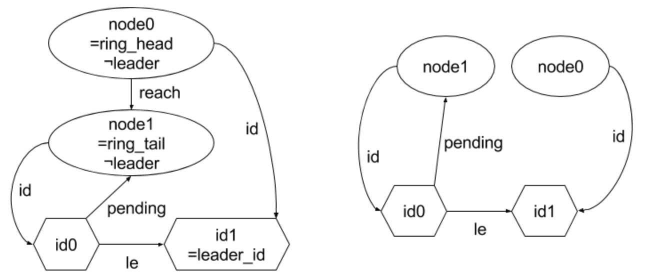

Figure 4. The first interaction step of Ivy on the leader protocol. The left graph shows a CTI. The right graph shows a generalization of the CTI obtained by the user's selection of the three relations *pending*, *le*, and *id*. The idea is that the topology and the leader are not relevant.

Generalization When a CTI represents an unreachable state, the user should strengthen the invariant by adding a new conjecture to eliminate the CTI. One possible conjecture is a universal formula which excludes the CTI and all the states which include it as a substructure [12]. However, usually in order to make progress we need to exclude many more states. This is where a user can provide feedback beyond automatic tools. The user selects a set of relevant relations and elements. This already defines a generalization of the CTI. Next, the `Test + Interpolate` procedure, applies bounded verification (with a user-specified bound) to verify the user's suggestion and generalize further. If the check succeeds (i.e., the bounded verification formula is unsatisfiable), Ivy defines a new conjecture (stronger than the user's suggestion) based on a minimal UNSAT core.

In the leader protocol, the user judges that the CTI of Figure 4(left) is unreachable, because there is a pending message to node1 with its own ID, despite the fact that node0 has a higher ID. Thus, the user generalizes away the irrelevant information, which results in the generalization depicted in Figure 4(right). Applying `Test + Interpolate` with bound 3 to this user-specified generalization succeeds and leads to the conjecture shown in It. 1 of Table 1. Three more steps of obtaining CTIs and generalizing them (not shown here), leads to the conjectures shown in It. 2-4. These conjectures form an inductive invariant for the leader election protocol.

2.4 Why EPR and Universal Invariants?

In principle, interactive strengthening can be applied whenever the problem of checking inductiveness is decidable. However, Ivy handles a rather limited case with two major limitations: (1) Conjectures can only include universal quantifications (2) RML is restricted in a way which guarantees that it is possible to check if an invariant is not-inductive using an EPR formula [19]. These are rather severe limitations but as we show, we are still able to model and verify with them interesting protocols. On the positive side, they lead to the following benefits:

- Every satisfiable EPR formula has a finite model represented as a graph which Ivy displays using dot.
- Tools like Iprover and Z3 can be used to check satisfiability. Indeed, the problem of SAT solving for EPR formulas is NEXPTIME complete and not undecidable as for the more general fragment solved by Z3. In particular, for EPR formulas, quantifier instantiation in Z3 no longer contains matching loops which can be annoying to users. In fact in our experience Z3 provides response time which is reasonable for an interactive mode.
- The bounded verification procedure shown in Figure 2 can be implemented w/o the need to a-priori bound the number of elements in the model.
- The `Find Minimal CTI` procedure can be effectively implemented using calls to the SAT solver as shown in Algorithm 1.
- The generalization procedure is implemented using UNSAT core which tools like Z3 already compute.
- Universal conjectures have a natural graphical representation which depicts the forbidden substructure.

3. RML: A Relational Modeling Language with Effectively Propositional Transitions

In this section we define a simple programming language, called Relational Modeling Language (RML), for modeling systems in a way that ensures that transitions can be described as formulas in EPR. We start by describing the syntax and informal semantics of RML. We then continue to define a weakest precondition operator for RML, and a large-step semantics in the form of a transition system.

3.1 RML Syntax and Informal Semantics

Figure 5 shows the abstract syntax of RML. RML imposes two main programming limitations: only a single loop is allowed and the only data structures are finite relations. Hence, an RML program is defined over a finite set of sorted variables and a finite set of finite relations, denoted V and R, respectively. Their values define the state of the program, where the value of a relation can be viewed as a finite table (or a finite set of tuples). Note that while the relations are always finite, there is no bound on the number of tuples in a relation, and thus an RML program has infinitely many states.

Assertions RML commands use assertions in many-sorted first-order logic over V and R. Formally, an RML program defines a vocabulary $\Sigma = V \cup R$, where every program variable $\mathbf{v} \in V$ is modeled as a *constant symbol* and every relation $\mathbf{r} \in R$ is modeled using a *relation symbol*. Assertions are closed first-order formulas over Σ (i.e., without any free logical variables). In the sequel, we use assertions and formulas interchangeably. Unless explicitly stated otherwise, we always refer to closed formulas.

It	Conj
0	$\big((\mathit{leader}(X) \land \mathit{leader}(Y)) \to X = Y)\big) \land \big(\mathit{leader}(X) \to \mathit{id}(X, \mathit{leader\_id})\big) \land \big((\mathit{id}(X, I) \land \mathit{id}(Y, J) \land \mathit{leader}(X)) \to \mathit{le}(J, I)\big)\big)$
1	$\neg(\mathit{id}(N_0, I_1) \land \mathit{id}(N_1, I_0) \land \mathit{le}(I_0, I_1) \land \mathit{pending}(I_0, N_1) \land I_0 \neq I_1)$
2	$\neg(\mathit{id}(N_0, I_1) \land \mathit{id}(N_1, I_2) \land \mathit{le}(I1, I2) \land \mathit{pending}(I_1, N_2) \land \mathit{reach}(N_0, N_1) \land \mathit{reach}(N_1, N_2) \land N_0 \neq N_1 \land N_2 \neq N_1)$
3	$\neg(\mathit{id}(N_0, I_2) \land \mathit{id}(N_2, I_1) \land \mathit{le}(I1, I2) \land \mathit{pending}(I1, N1) \land \mathit{reach}(N_0, N_1) \land \mathit{reach}(N_1, N_2) \land N_1 \neq N_0)$
4	$\neg(\mathit{id}(N_1, I_1) \land \mathit{id}(N_2, I_2) \land \mathit{le}(I_1, I_2) \land \mathit{pending}(I_1, N_0) \land \mathit{reach}(N_0, N_1) \land \mathit{reach}(N_1, N_2) \land N_1 \neq N_2)$

Table 1. The successive conjuncture interactions in the leader protocol. Capitalized variables are universally quantified. The first conjunctures (It 0) is the safety property and the last conjuncture (It 4) is the resulting inductive invariant. Between iterations the user applied generalization using `Find Minimal CTI` and `Test + Interpolate`.

$\langle \mathit{rml} \rangle$	::=	$\langle \mathit{decl} \rangle$; `assume` $\varphi_{\texttt{EA}}$; `while` $*$ `do` $\langle \mathit{cmd} \rangle$; $\langle \mathit{cmd} \rangle$	
$\langle \mathit{cmd} \rangle$	::=	`skip`	do-nothing
	$\mid$	`abort`	terminate-abnormally
	$\mid$	$\mathbf{r}(\overline{x}) := \varphi_{\texttt{QF}}(\overline{x})$	quantifier-free update of **r**
	$\mid$	$\mathbf{v} := *$	havoc of variable **v**
	$\mid$	`assume` $\varphi_{\texttt{EA}}$	assumption
	$\mid$	$\langle \mathit{cmd} \rangle$; $\langle \mathit{cmd} \rangle$	sequential composition
	$\mid$	$\langle \mathit{cmd} \rangle \mid \langle \mathit{cmd} \rangle$	nondeterministic choice
$\langle \mathit{decl} \rangle$	::=	ϵ $\mid$ `sort` **sort** ; $\langle \mathit{decl} \rangle$	
	$\mid$	`var` **v** : **sort** ; $\langle \mathit{decl} \rangle$ $\mid$ `rel` **r** : $\overline{\textbf{sort}}$; $\langle \mathit{decl} \rangle$	

Figure 5. Syntax of RML. **r** denotes an n-ary relation, $\overline{x}$ denotes a vector of n logical variables, **v** denotes a variable, **sort** denotes a sort and $\overline{\textbf{sort}}$ denotes a vector of sorts separated by commas. $\varphi_{\texttt{QF}}(\overline{x})$ denotes a quantifier-free formula with free logical variables $\overline{x}$, and $\varphi_{\texttt{EA}}$ denotes a formula with quantifier prefix $\exists^*\forall^*$.

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

Programs An RML program has the form

Decls ; `assume` $\varphi_{\texttt{init}}$; `while` $*$ `do` $C_{\texttt{body}}$; $C_{\texttt{final}}$. The program starts with a sequence of declarations of the program variables and relations, followed by an assumption on their initial values provided by a $\exists^*\forall^*$-formula. These define the sets V and R and the initial states of the program: every state that satisfies $\varphi_{\texttt{init}}$ is a possible initial state of the program. The core of the program is a nondeterministic `while` loop. In each iteration, the program either executes the command $C_{\texttt{body}}$ or exits the loop. When the loop terminates, the $C_{\texttt{final}}$ command is executed.

In the sequel, we fix an RML program, denoted *Prog*, defined as above.

Commands Each command investigates and potentially updates the state of the program. The semantics of `skip` and `abort` are standard. The command $\mathbf{r}(x_1, \ldots, x_n) := \varphi_{\texttt{QF}}(x_1, \ldots, x_n))$ is used to update the n-ary relation **r** to the set of all n-tuples that satisfy the quantifier-free formula $\varphi_{\texttt{QF}}$. For example, $\mathbf{r}(x_1, x_2) := (x_1 = x_2)$ updates the binary relation **r** to the identity relation. $\mathbf{r}(x_1, x_2) := \mathbf{r}(x_2, x_1)$ updates **r** to its inverse relation. The havoc command $\mathbf{v} := *$ performs a nondeterministic assignment to **v**. The `assume` command is used to restrict the executions of the program to those that satisfy the given $\exists^*\forall^*$-formula. Finally, sequential composition and nondeterministic choice are defined in the usual way. Table 2 provides several useful syntactic sugars, including a `if-then-else` command and `insert` and `remove` commands for relations.

Syntactic Sugar	Command
`assert` $\varphi_{\texttt{AE}}$	$\{$`assume` $\neg\varphi_{\texttt{AE}}$; `abort`$\} \mid$ `skip`
`if` $\varphi_{\texttt{AF}}$ `then` C_1 `else` C_2	$\{$`assume` $\varphi_{\texttt{AF}}$; $C_1\} \mid \{$`assume` $\neg\varphi_{\texttt{AF}}$; $C_2\}$
$\mathbf{r}$.`insert`$(\overline{x} : \varphi_{QF}(\overline{x}))$	$\mathbf{r}(\overline{x}) := \mathbf{r}(\overline{x}) \lor \varphi_{QF}(\overline{x})$
$\mathbf{r}$.`remove`$(\overline{x} : \varphi_{QF}(\overline{x}))$	$\mathbf{r}(\overline{x}) := \mathbf{r}(\overline{x}) \land \neg\varphi_{QF}(\overline{x})$
$\mathbf{r}$.`insert`$(\overline{\mathbf{v}})$	$\mathbf{r}(\overline{x}) := \mathbf{r}(\overline{x}) \lor (\overline{x} = \overline{\mathbf{v}})$
$\mathbf{r}$.`remove`$(\overline{\mathbf{v}})$	$\mathbf{r}(\overline{x}) := \mathbf{r}(\overline{x}) \land \neg(\overline{x} = \overline{\mathbf{v}})$

Table 2. Syntactic sugars for RML. $\varphi_{\texttt{AE}}$ denotes a formula with $\forall^*\exists^*$ prefix. $\varphi_{\texttt{AF}}$ denotes an alternation-free formula. **r** denotes an n-ary relation, $\overline{x}$ denotes a vector of n logical variables, and $\overline{\mathbf{v}}$ denotes a vector of n program variables.

Safety properties The `abort` and the `assert` commands are used to specify error traces for the program: An error trace is an execution that leads to an abort or assertion violation. A program is safe if it has no error trace.

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 first order logic. We start with a formal definition of the set of states as structures of first order logic.

States Recall that a state of an RML program consists of a valuation of all the program variables and all the relations. Formally, a state of the program is given by a first-order structure over Σ, defined as follows. (Recall that program variable are constants in Σ).

Definition 1 (Structures)**.** Given a vocabulary Σ, a *structure* s of Σ is a pair $s = (D, \mathcal{I})$, where D is its *finite* many-sorted domain, and $\mathcal{I}$ is its 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 $\mathcal{I}(r) : D^k \to \{0, 1\}$, and associates each constant symbol $c \in \Sigma$ with a function $\mathcal{I}(c) : D \to \{0, 1\}$, such that $\mathcal{I}(c)(e) = 1$ for exactly one element $e \in D$. [1] $\mathcal{I}$ obeys the sort restrictions.

[1] For uniformity of the presentation of our approach, we treat constants as "special" unary relations which consist of exactly one element.

$$\begin{array}{rcl}
wp(\texttt{skip}, Q) & = & Q \\
wp(\texttt{abort}, Q) & = & \mathit{false} \\
wp(\mathbf{r}(\overline{x}) := \varphi_{\text{QF}}(\overline{x}), Q) & = & Q[\varphi_{\text{QF}}(\overline{t})/\mathbf{r}(\overline{t})] \\
wp(\mathbf{v} := *, Q) & = & \forall x.\, Q[x/\mathbf{v}] \\
wp(\texttt{assume}\ \varphi_{\text{EA}}, Q) & = & \varphi_{\text{EA}} \rightarrow Q \\
wp(C_1; C_2, Q) & = & wp(C_1, wp(C_2, Q)) \\
wp(C_1 | C_2, Q) & = & wp(C_1, Q) \wedge wp(C_2, Q)
\end{array}$$

Figure 6. Rules for wp. $Q[y/x]$ denotes Q with occurrences of x substituted by y.

Note that we consider only structures over a finite domain (hence, every relation corresponds to a finite table).

Weakest precondition of RML commands Figure 6 presents the definition of a *weakest precondition* operator for RML, denoted wp. The weakest precondition of a command C with respect to an assertion Q, denoted $wp(C, Q)$, is 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. Importantly, $\forall^*\exists^*$-formulas are closed under wp:

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

Safety Formally, an RML program *Prog* is *safe* if for every $k \geq 0$, $\varphi_{\texttt{init}} \Rightarrow wp(C^k_{\texttt{body}}\ ;\ C_{\texttt{final}}, \mathit{true})$, where $C^0_{\texttt{body}} \stackrel{\text{def}}{=} \texttt{skip}$ and $C^{i+1}_{\texttt{body}} \stackrel{\text{def}}{=} C^i_{\texttt{body}}\ ;\ C_{\texttt{body}}$.

3.3 From RML Programs to EPR Transition Systems

In this section we define the semantics of an RML program as a transition system expressed in EPR. In the following sections we utilize special properties of EPR transition systems to develop an interactive verification framework for RML programs.

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 relational vocabulary that may contain constant symbols and relation symbols but no function symbols) with a quantifier prefix $\exists^*\forall^*$ when written in prenex normal form (i.e., when written as a prefix of quantifiers followed by a quantifier-free formula). Satisfiability of EPR formulas is reducible to Boolean SAT, hence decidable [10]. Moreover, formulas in this fragment enjoy the *small model property*, meaning that a satisfiable formula is guaranteed to have a finite model of a size proportional to the depth of quantifier nesting. As a result, checking satisfiability of EPR formulas is NEXPTIME-complete.

From RML commands to EPR transitions Each command in an RML program represents a set of *transitions* between the state before and the state after the command. Formally, sets of transitions are described using transition formulas. These are *two-vocabulary* closed formulas, over vocabulary $\Sigma \cup \Sigma'$ where $\Sigma' = \{a' \mid a \in \Sigma\}$. A two-vocabulary transition formula τ, represents the set of all transitions from $s = (D, \mathcal{I})$ to $s' = (D, \mathcal{I}')$ such that $(D, \mathcal{I} \cup \mathcal{I}'') \models \tau$, where $\mathcal{I}''$ interprets Σ' by $\mathcal{I}''(a') = \mathcal{I}'(a)$. For s and s' as above, we write that $(s, s') \models \tau$.

The transition formula τ_C for command C is $\tau_C \stackrel{\text{def}}{=} \neg wp(C^s, \neg id)$ where C^s is obtained from C when substituting every `abort` command by `skip`, and id is a two-vocabulary formula that specifies that the input and the output states are identical, i.e., $id \stackrel{\text{def}}{=} \bigwedge_{v \in V} v = v' \wedge \bigwedge_{r \in R} \forall \overline{x} : r(\overline{x}) \leftrightarrow r'(\overline{x})$. Note that τ_C is closed and defined over a two-vocabulary alphabet $\Sigma \cup \Sigma'$. Note further that since $\neg id$ is a $\exists^*$-formula, and since $\forall^*\exists^*$ formulas are closed under wp (Lemma 3.2), then $wp(C^s, \neg id)$ is a $\forall^*\exists^*$-formula, making the transition formula a $\exists^*\forall^*$ formula (i.e., in EPR).

Lemma 3.3. *Let C be an RML command. Then τ_C is an EPR formula.*

An RML program naturally defines a transition system:

Definition 2 (Transition System of *Prog*)**.** The *transition system* associated with an RML program *Prog* is $\mathit{TS} = (\varphi_0, \tau, \varphi_P)$, where $\varphi_0 = \varphi_{\texttt{init}}$ is the initial condition formula, $\tau = \tau_{C_{\texttt{body}}}$ is the transition relation formula, and $\varphi_P = wp(C_{\texttt{body}} | C_{\texttt{final}}, \mathit{true})$ is the safety property formula.

The initial condition of *TS* describes the states that are reachable at the loop entry point. Each transition describes one execution of the loop body, and the safety property describes the states that when occur at the head of the loop do not lead to any abort, neither within the loop body $C_{\texttt{body}}$, nor in the epilog $C_{\texttt{final}}$. Reachable states of *TS* are defined in the usual way. The definition of *TS* ensures the following:

Lemma 3.4. *Let Prog be an RML program and TS the transition system associated with it. Then Prog is safe if and only if all the reachable states of TS satisfy φ_P.*

Note that φ_0 and φ_P are defined over the vocabulary Σ, and τ is defined over $\Sigma \cup \Sigma'$. All formulas are closed. The following definition and lemma are essential for the ability to verify RML programs:

Definition 3 (EPR transition system)**.** $\mathit{TS} = (\varphi_0, \tau, \varphi_P)$ is an *EPR transition system* if φ_0 and τ are $\exists^*\forall^*$-formulas, and φ_P is a $\forall^*\exists^*$-formula.

Lemma 3.5. *Let $\mathit{TS} = (\varphi_0, \tau, \varphi_P)$ be the transition system associated with an RML program. Then TS is an EPR transition system.*

4. Model Debugging via Symbolic Bounded Verification

In this section we show that RML programs, due to their formulation as EPR transition systems, are amenable to *bounded verification*, where the number of executions of the loop is bounded, but the state of the program remains *unbounded*. Namely, the verification applies to an *unbounded* state space, in contrast to traditional bounded model checking approaches. Bounded verification has many applications. One of them is the debugging of system models given by RML programs, as depicted in Figure 2 (see Section 2.2).

Bounded verification A transition system *TS* $= (\varphi_0, \tau, \varphi_P)$ is k-*safe* if all the states of *TS* that are reachable in at most k τ-transitions satisfy φ_P. A conjecture φ is a k-*invariant* of *TS* if *TS*$' = (\varphi_0, \tau, \varphi)$ is k-safe. If *TS* is obtained from an RML program, k-bounded safety implies that *any* execution of the corresponding RML program where the loop is executed at most k times, staring from *any* possible initial state, is safe (i.e., has no abort or assertion violation).

k-bounded safety holds for *TS* iff the formula $\mathrm{BMC}_n(\mathit{TS})$ presented next is unsatisfiable for every $0 \leq n \leq k$. Hence, k-bounded verification amounts to checking (un)satisfiability of these formulas.

$$\mathrm{BMC}_n(\mathit{TS}) = \varphi_0(\Sigma_0) \wedge \bigwedge_{i=0}^{n-1} \tau(\Sigma_i, \Sigma_{i+1}) \wedge \neg\varphi_P(\Sigma_n) \quad (1)$$

where $\Sigma_i = \{a_i \mid a \in \Sigma\}$. $\mathrm{BMC}_n(\mathit{TS})$ is therefore a formula over vocabulary $\bigcup_{i=0}^{n} \Sigma_n$, which consists of $n+1$ copies of Σ. Further, $\mathrm{BMC}_n(\mathit{TS})$ is an EPR formula. This implies the decidability of bounded verification of EPR transition systems (and RML programs):

Theorem 4.1. *Given an EPR transition system TS, and a bound $k \geq 0$, k-bounded verification amounts to checking unsatisfiability of EPR formulas, hence it is decidable.*

Counterexamples If bounded verification fails, i.e. $\mathrm{BMC}_k(\mathit{TS})$ is satisfiable, then a model of $\mathrm{BMC}_k(\mathit{TS})$ defines a *counterexample* in the form of a sequence of states $s_0, \ldots, s_k$, such that s_0 is an initial state of *TS* (i.e., $s_0 \models \varphi_0$), s_k is a "bad" state (i.e., $s_k \models \neg\varphi_P$), and there is a transition between every two consecutive states (i.e., $(s_i, s_{i+1}) \models \tau$ for every $0 \leq i < k$).

5. An Interactive Methodology for Obtaining Universal Inductive Invariants

In this section, we describe our interactive approach for *unbounded verification* of EPR transition systems, such as the ones obtained from RML programs. The idea is to assist a user in the construction of universal inductive invariants.

5.1 Inductive Invariants for EPR Transition Systems

A closed formula I over Σ is an *inductive invariant* for a transition system *TS* $= (\varphi_0, \tau, \varphi_P)$ if the following conditions hold: (i) (initiation) $\varphi_0 \Rightarrow I$, (ii) (safety) $I \Rightarrow \varphi_P$, and (iii) (consecution) $I \wedge \tau \Rightarrow I'$, where I' denotes the formula over Σ' obtained from I when every symbol $a \in \Sigma$ is replaced by a'. If $s \models \varphi_0 \wedge \neg I$, or $s \models I \wedge \neg\varphi_P$, or $(s, s') \models I \wedge \tau \wedge \neg I'$, then s is a *counterexample to induction (CTI)*.

An inductive invariant of *TS* is in particular an invariant, i.e. it holds in all the reachable states of *TS*. Thus, if there exists an inductive invariant for a transition system *TS*, then all the reachable states of *TS* satisfy φ_P.

The following theorem implies that for EPR transition systems, it is decidable to determine if a given universal formula I is an inductive invariant.

Theorem 5.1. *Given an EPR transition system TS and an alternation-free formula I, checking whether I is an inductive invariant for TS amounts to checking unsatisfiability of EPR formulas, hence it is decidable.*

5.2 Overview of the Interactive Search for Universal Inductive Invariants

Consider an EPR transition system *TS* $= (\varphi_0, \tau, \varphi_P)$. Figure 3 presents our methodology for obtaining a universal inductive invariant I for *TS* as a conjunction of *conjectures*, i.e., $I = \bigwedge_{i=1}^{k} \varphi_i$. Each conjecture φ_i is a closed universal clause, i.e. a disjunction of literals with a $\forall^*$ prefix. In the sequel, we interchangeably refer to I as a set of conjectures and as the formula obtained from their conjunction.

Our approach strongly depends on the following three properties of EPR transition systems and universal inductive invariants. (i) decidability of bounded verification (Theorem 4.1), (ii) decidability of checking inductiveness (Theorem 5.1), and (iii) the ability to graphically present universal conjectures in a way that is intuitive for system developers, as we elaborate in Section 5.4.

Our methodology guides a user through an iterative search for a universal inductive invariant I. The search starts from a given set (conjunction) of conjectures as a candidate inductive invariant I. For example, I can be initialized to *true*. If φ_P is universal, then I can also be initialized to φ_P. If the search starts after a modification of the model, then conjectures that were learnt before can be reused. Additional initial conjectures can be computed by applying basic abstract interpretation techniques.

Each iteration first automatically checks whether the current candidate I is an inductive invariant. If I is not yet an inductive invariant, a CTI is presented to the user. The user can either strengthen I by conjoining it with an additional conjecture, or weaken I by eliminating one (or more) of the conjectures. The user can also choose to modify the model in case the CTI indicates a bug in the model. This process continues until an inductive invariant is found.

Stregtening/Weakening For I to be an inductive invariant, all the conjectures φ_i need to be invariants (i.e., hold in all the reachable states). This might not hold in the intermediate steps of the search, but it guides strengthening and weakening: strengthening aims at only adding invariants as conjectures, and weakening aims at identifying and removing conjectures that are not invariants.

While strengthening and weakening are ultimately performed by the user, we provide the user with assistance in the form of automated checks:

Obtaining a minimal CTI If I is not inductive, then a counterexample to induction can be automatically obtained from the failed inductiveness check. It is desirable to present the user with a CTI that is easy to understand, and not cluttered with many unnecessary features. To this end, we automatically search for a "minimal" CTI, where the minimization parameters are defined by the user (see Section 5.3).

Interactive generalization To eliminate the CTI, the user needs to either strengthen I to exclude s from I, or to weaken I to include s (if $s \models \varphi_0 \wedge \neg I$) or its successor state s' if $(s, s') \models I \wedge \tau \wedge \neg I'$. Intuitively, if the CTI 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.

Generalizations are explained in Section 5.4, and the interactive generalization process is explained in Section 5.5. Here again, the user defines various parameters for generalization, and the tool 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 5.2. If all the conjectures added by the user exclude only unreachable states (i.e., all are invariants), then weakening is never required. 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.

5.3 Obtaining Minimal CTIs

We refine the search for CTIs by trying to find a minimal CTIs according to heuristic measures. As a general rule, smaller CTIs are desirable since they are both easier to understand, which is important for our 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. As such, we define a set of 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: $\exists \overline{x_1}, \ldots \overline{x_n}. \forall \overline{y}. \left(r(\overline{y}) \to \bigvee_{i=1}^{n} \overline{y} = \overline{x_i}\right)$, where $\overline{x_1}, \ldots, \overline{x_n}, \overline{y}$ denote k-tuples of logical variables.

Algorithm 1: Obtaining a Minimal CTI

```
1 if $\psi_{cti}$ 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             $\psi_{min} := \psi_{min} \wedge \varphi_m(n)$ ; break ;
7 let $(s, s')$ be such that $(s, s') \models \psi_{cti} \wedge \psi_{min}$ ;
8 return $s$
```

5.4 Partial Structures as Generalizations

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 and constant symbols are interpreted as *partial* functions. Formally:

Definition 4 (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 constant symbol $c \in \Sigma$ with a partial function $\mathcal{I}(c) : D \rightharpoonup \{0, 1\}$, such that $\mathcal{I}(c)(e) = 1$ for at most one element $e \in D$.

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. Informally, a partial structure generalizes a CTI (structure) by turning some values ("facts") to undefined. 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 partial structure is still unreachable, regardless of the value of $\mathcal{I}(r)(\overline{e})$. Formally, we define a *generalization partial order* over partial structures:

Definition 5 (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/constant 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.

For a structure s of Σ over D, we say that a partial structure s' is a *generalization* of s if $s' \sqsubseteq s$. When s_1 and s_2 are generalizations of a CTI, $s_2 \sqsubseteq s_1$ means that s_2 is more general (represents more states) than s_1 (s_1 is more specific).

From partial structures to conjectures Every partial structure (generalization) induces a (universal) conjecture that excludes the set of states that it represents. Technically, the conjecture associated with a partial structure is the universal formula equivalent to 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 6 (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) constant or relation symbol $a \in \Sigma$ such that e_i appears in the domain of $\mathcal{I}(a)$. The *diagram* of s, denoted by *Diag*(s), is the following formula over Σ:

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

where ψ is the conjunction of:

- $x_i = c$ for every constant c such that $\mathcal{I}(c)(e_i) = 1$, and
- $x_i \neq c$ for every constant c such that $\mathcal{I}(c)(e_i) = 0$, and
- $r(x_{i_1}, \ldots, x_{i_k})$ 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. $\mathcal{I}(r)(e_{i_1}, \ldots, e_{i_k}) = 0$.

Intuitively, one can think of *Diag*(s) as the formula produced by treating individuals in D as existentially quantified variables and explicitly encoding the interpretation of every constant and every relation symbol for which it is defined. The diagram is well defined since we consider *finite* structures.

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 7 (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 5.3. *Let s be a partial structure and let s' be a (total) structure such that $s \sqsubseteq s'$. Then $s' \not\models \varphi(s)$.*

Therefore, by using the graphical interface to define partial structures, the user is also able to define new conjectures. 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.

5.5 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 $\mathcal{I}(r)(\overline{e})$ is undefined, she can use this intuition to define the upper bound.

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

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

If verification fails, the user is presented with a trace that explains the violation. Based on this trace, the user can either redefine the generalization range, can decide to weaken I or can decide to modify the model.

If bounded verification succeeds, 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', as desired. The partial structure s' is displayed to the user as a candidate generalization. (The user can also see the corresponding conjecture.)

User investigates the suggested generalization: In order to decide whether to accept a suggested generalization, the user can check additional properties of the generalization (and the conjecture associated with it). For example, the user may check whether it is unreachable also in k' steps, for some $k' > k$. During this phase the user may also manually change the obtained conjecture (generalization) by reintroducing interpretations that became undefined. Finally, the user decides whether to keep the conjecture, or try to obtain a different generalization by changing her choices.

6. 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 of the procedures described in Section 4 and Section 5. Ivy provides a graphical user interface implemented using JavaScript in an IPython [18] notebook. The supplementary material contains sourcec files for the presented protocols.

6.1 Protocols

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 models the reflexive transitive closure of *route*. The modeling maintains *route** using the standard technique for updating transitive closure (e.g. [10]). 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., [22]) has provided a chain based mechanism to provide these guarantees in multinode 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 [21], 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 as similar to Houdini [5] in that we described a class of formulas using a template, and used abstract interpretation 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 inductive invariant in EPR.

6.2 Results & Discussion

Next, we evaluate Ivy's effectiveness. We begin, in Table 3 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 relations, 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.

Finally, in some cases the inductive invariants found by Ivy were too strong, and indicated we needed to further refine our model. For example, the initial set of inductive invariants we found for the database example did not depend on transactions aborting or committing, which was a result of our model being too week. Once fixed, we could reuse work previously done in finding inductive invariants.

Overall Thoughts Overall we found that Ivy makes it much easier for users to actually find inductive invariants, and provides a guided experience through this process. This is in contrast to the existing model for finding such invariants, where users must come up with invariants by manual reasoning, and have little feedback other than whether their invariant is inductive or not.

7. 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 [8]. This approach has been generalized in the STRAND logic [16]. 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 countermodel 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.

Protocol	**S**	**R**	**P**	**I**	**G**
Leader election in ring	2	5	9	36	4
Learning switch	2	6	6	20	5
Database chain replication	4	13	14	42	8
Chord ring maintenance	1	16	35	47	4

Table 3. Protocols verified interactively with Ivy. **S** is the number of sorts in the model. **R** is the number of relations in the model. **P** is the size of the initial set of conjectures, measured by the total number of literals that appear in the formula. **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.

Various techniques have been proposed for solving the paramaterized model checking problem (PMCP). Some achieve decidability by restricting the process model to specialized classes that have cutoff results [6] 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 [20]. 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. In practice it is quite difficult to produce proofs the kinds of systems described here.

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 [12]. 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 ecode all of methematics but provide little automation, and tools such as Dafny [14] 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 countermodels, and in the case of a true counterexample to induction they provide little feedback as to the root cause of the proof failure.

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 R. Jhala and D. A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87. Springer, 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] 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, Berlin, Germany, March 12-16, 2001, Proceedings*, pages 500–517, 2001.

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

[7] 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 2015, Monterey, CA, USA, October 4-7, 2015*, pages 1–17, 2015.

[8] 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 E. Brinksma, R. Cleaveland, K. G. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for Construction and Analysis of Systems, First International Workshop, TACAS '95, Aarhus, Denmark, May 19-20, 1995, Proceedings*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110. Springer, 1995.

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

[10] 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 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 756–772, 2013.

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

[12] 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 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 583–602, 2015.

[13] 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.

[14] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In E. M. Clarke and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16, Dakar, Senegal, April 25-May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

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

[16] P. Madhusudan and X. Qiu. Efficient decision procedures for heaps using STRAND. In E. Yahav, editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2011.

[17] 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.

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

[19] 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.

[20] A. Pnueli, S. Ruah, and L. D. Zuck. Automatic deductive verification with invisible invariants. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2001.

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

[22] 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.