Abduction by Non-Experts

Nikolaj Bjørner[2], Dejan Jovanović[1], Tancrède Lepoint[1], Philipp Rümmer[3], and Martin Schäf[1]

[1] SRI International
[2] Microsoft Research
[3] Uppsala University

Abstract

Crowdsourcing promises to quasi-automate tasks that cannot be automated otherwise. Success stories like natural language translation or recognition of cats in images show that carefully crafted crowdsourcing tasks solve large problem instances which could not be solved otherwise. To utilize crowdsourcing, one has to define the problem in a way that is easy to split into small tasks, that the tasks are easy to solve for humans and hard to solve for a machine, and that the machine can efficiently check if the solution is correct.

In this paper we discuss a novel approach of using crowdsourcing to assist software verification. We argue that Horn clauses form a good base for crowdsourcing since they are easy to subdivide, and that logic abduction is a suitable task since it is hard to automatically find abductive inferences for Horn clauses, but it is easy to check if an inference makes a Horn clause valid. We describe a prototype implementation, we show how crowdsourcing integrates in the verification process, and present preliminary results.

1 Introduction

Crowdsourcing is becoming a popular tool in different areas of computer science to achieve quasi-automation of complex problems that cannot be automated otherwise. Problems like image recognition, translation of texts into foreign languages, or creation of maps are just a few examples that successfully rely crowdsourcing. Lately, we see more and more interest in crowdsourcing from the software engineering community [22]. In this paper, we want to discuss the merits of crowdsourcing software verification. While automation of software verification tools is improving, occasional manual interaction is inevitable, and to make verification really useful in practice, a lot of legacy code (like libraries, device drivers, etc), needs to be verified by someone. These verification tasks are not overly time critical, so crowdsourcing them seems like a feasible solution. In this paper we discuss how such a crowdsourcing approach could be implemented and discuss a prototype.

There are many different approaches to crowdsourcing. Citizen science and gamification-based approaches such as SETI@home [1] or FoldIt [10] have received a lot of attention. For verification, however, these approaches may not be the most suitable since the task at hand requires a certain amount of expertise. For this paper, we envision drawing a userbase from platforms like Amazon mechanical turk, or Upwork, where certain requirements can be enforced on the background of the crowd users.

Crowdsourcing of verification has caught some interest from the community in the recent years. DARPA ran a project on crowdsourcing verification problems with several teams developing games that crowdsource different aspects of various verification approaches. For example, one of the developed games, GhostMap [23, 30], crowdsources the problem of finding suitable refinements in a CEGAR-based model checker. Another game, Xylem [13], crowdsources predicate discovery to assist the abstract interpretation tool Frama-C. Other tools, like Binary-Fission [12] crowdsource invariant discovery to obtain more readable invariants than existing machine learning approaches, and Paradox or Pipejam [8] turn constraint systems from type checking into puzzle games.

A survey [7] discusses a set of common issues that the teams in the program faced. One of the major challenges described is the ability to control the difficulty of the crowdsourced problems. On one hand, the problems are expected to be complex since they could not be solved without crowdsourcing, on the other hand they have to be simple enough be solvable by a human in a reasonable time. Further, the problem has to be formulated in a way that checking the solution is cheap, otherwise we cannot give feedback to users. This is of particular importance for mechanical turk based crowdsourcing where users get paid for correct answers. For solutions that gamify the crowdsourcing, an additional requirement is to get a notion of difficulty of problems. If the difficulty of problems changes randomly, it is hard to develop a game that keeps users engaged over a long period of time.

We summarize these problems in the following research questions that we want to address in this paper. How can we build a crowd-sourcing infrastructure for software verification where:

1. The crowdsourced problems are of a nature where human creativity can out-perform the machine.
2. Crowdsourcing problems can be generated from failed verification attempts and the complexity of the crowdsourced problem can be parameterized.
3. User provided solutions are cheap to check.

To answers these questions we propose an implementation based on Constrained Horn Clauses (CHC) and logical abduction. CHCs provide a convenient way to encode a variety of software verification problems [3, 4, 15]. In this encoding, the set of possible states at a particular program location is represented by a predicate over all program variables that are live at this location (e.g., $p(x, y, z)$) and each transition between locations is represented by a Horn clause. For example, a clause $p(x, y, z + 1) \leftarrow q(x, y, z) \wedge x < y$ represents a transition where z gets incremented by one if $x < y$. Tools like SeaHorn [17] and JayHorn [18] can generate such CHCs from C or Java programs.

A Horn solver, like HSF [16], Eldarica [28], Spacer [19], or Z3 [6] solves a system of CHCs by finding a formula for each predicate such that all Horn clauses become valid (or a counter example can be found). The solver finds these formulas by either starting from the precondition (i.e., the Horn clauses that have no predicate in the body), or from the assertions (the Horn clauses that have no predicate in the head) and propagating results until either a solution is found or the current assignments create a contradiction, in which case Craig interpolation can be used to refine previous assignments. A Horn solver may get stuck if all predicates are assigned and there is no contradiction, but some Horn clauses are only Sat but not valid. In this case, we need logic abduction to either strengthen a predicate in the body or weaken the predicate in the head of the Horn clause.[1] *This is the part of verification that we want to crowdsource.*

Previous verification approaches using abductive inference (e.g., [9, 29]) have shown that the technique is powerful, but that it is hard to find right the abductions automatically since there is little information to learn from in the Horn clauses. This addresses our first research question: we hypothesize that humans are generally better at making abductive inferences than machines because they can draw from intuition and experience to chose formula expressions that may be suitable. While we could equip a solver with a portfolio of useful patterns to find a good inference, we can always find (or construct) an abduction problem where the patterns available to the tool are not sufficient.

[1]Logic abduction is the problem where, given an implication $A \to B$ that is but not valid, we want to find a C such that $A \wedge C$ is satisfiable and $C \wedge A \to B$ is valid.

To generate such an abduction problem for crowdsourcing, we run a Horn solver on a verification problem as discussed above until it gets stuck at a point where at least one clause is only Sat but not Valid. This clause together with the current assignments to the predicates is then turned into a crowdsourcing problem. We can now tune the difficulty of the problem by choosing how many other clauses that share predicates will be included. The more Horn clauses we include, the harder it becomes to find an assignment that makes all clauses valid. This addresses our second research question and gives us fine grained control over the difficulty of the problem.

The third research question is also addressed by constructing problems this way. Since a solution to a crowdsourced problem is an assignment to the predicates, we can simply instantiate the predicates for each Horn clause and use an SMT solver to check if they are valid. This check is sufficiently fast to provide immediate feedback to the user.

In Section 2 we walk through the crowdsourcing process using a small example. In Section 3 we formalize the approach. Section 4 outlines the implementation of our prototype and sketches the directions for the future, and Section 5 shows discusses some first results.

Contributions. The main contribution of this paper is to show that the Horn clause formalism used in software verification translates nicely into crowdsourcing terminology. Horn clauses can be split into self contained sub-problems. Improving predicate assignments with abduction is a task where human experience potentially can have an edge over algorithms, and checking if a user-provided solution is a valid abduction can be done efficiently.

We do not claim any silver bullet or magic potion: this crowdsourcing technique only helps if there exists a solution to the Horn encoding that can be checked by an SMT solver. It may still be the case that there exists a solution but that this solution is not expressible in a decidable theory. In this case, our approach will not help either. Further, the details of how to split Horn clauses into sub-problems and how to present problems to the user can be implemented in many different ways and we do not argue that our way is better than others.

The second contribution is an web-based implementation to crowdsource Horn solving. The system is online to play with and can easily be adopted to use different ways of splitting CHCs and different user interfaces.

2 Example

We illustrate how crowdsourcing can help Horn solvers to solve software verification tasks by providing abductive inferences using the example program in Figure 1. Note that many state of the art Horn solvers solve this problem without the help of abduction due to various optimizations but it is suitable to motivate our approach.

The curious reader can try the system on a set of benchmark problems online immediately at `http://www.horn-abduction.org`.

Let us assume we want to verify the assertion in line 7 and we agree to ignore potential numeric overflow bugs. Then we can use a tool like SeaHorn to obtain a system of CHCs that models this verification condition that look similar to the clauses in Figure 2. In practice, these CHCs tend to be more complex because they have to encode various assumptions and initialization code.

Each of the predicate p_0, p_1, and p_2 represents the set of possible states at a particular program location. The predicate p_0 summarizes the possible states at the loop head in line 3, p_1 summarizes the states after the execution of the loop body, and p_2 the states after exiting

```
1 int x = 0;
2 int y = 0;
3 while (x < n) {
4   x = x + 1;
5   y = y + 2;
6 }
7 assert(x + y == 3 * n);
```

Figure 1: Example program to illustrate how abduction can assist Horn solving. The program increments x by one and y by two until x becomes bigger than n and then asserts that the sum of x and y is three times n.

$$\begin{aligned} p_0(0,0,n) &\leftarrow n \geq 0 && (1)\\ p_1(x,y,n) &\leftarrow p_0(x,y,n), x < n && (2)\\ p_2(x,y,n) &\leftarrow p_0(x,y,n), x \geq n && (3)\\ p_0(x+1,y+2,n) &\leftarrow p_1(x,y,n) && (4)\\ x+y=3n &\leftarrow p_2(x,y,n) && (5) \end{aligned}$$

Figure 2: Horn encoding of the program in Figure 1. The predicate p_0 corresponds to the states at line 3, p_1 represents the states after the loop body at line 6, and p_2 the states before the execution of the assertion in line 7.

the loop (and before executing the assertion) line 7. For a Horn clause $H \leftarrow B$, we call B the body and H the head. The intuition is that if the body holds we can move to the head.

The first Horn clause describes the initialization and the implicit precondition that n is non-negative. This clause has no predicate in the body and is thus referred to as *fact*. The second Horn clause is the condition to enter the loop. It describes the effect of executing the loop body. The third clause is the condition for leaving the loop. The last Horn clause represents the assertion. Clauses that have no predicate in the head are referred to as *query*.

Let us assume our Horn solver applies a property directed algorithm that starts from the query and works backwards. Initially all predicates are assigned to the trivial solution *true*. For this assignment, all Horn clauses but the last one are valid. In the first iteration, the algorithm will update p_2 to the formula $x+y=3n$ from the head of the last Horn clause. This update validates (5). Now, the only non-valid clause is (3): $x+y=3n \leftarrow true, x \geq n$ which causes the algorithm to update p_0 to $x+y=3n \wedge x \geq n$. Next, we update p_1 to $x+1+y+2=3n \wedge x+1 \geq n$ based on Horn clause (4). With that assignment, the second clause becomes unsatisfiable and we can use Craig interpolation to improve p_0.

Now, all Horn clauses but (1) are valid. For simplicity, let us assume that the Horn solver doesn't know any tricks to recover from this (in practice they do, but they may up in the same situation later). The problem that we want to crowd source is to find a better assignment for p_0 that validates the Horn clause:

$$p_0(0,0,n) \leftarrow n \geq 0$$

Searching this assignment can be seen as finding an abduction to refine p_0.

We can now chose the difficulty of the crowdsourcing problem by deciding how many clauses we want to show to the user. If we only show this one clause it is relatively simple to find a solution that makes the formula valid (e.g., $p_0 := n \geq 0$). To increase the difficulty, we can also show other Horn clauses that contain p_0 and are valid under the current assignment to p_0. For example:

$$p_0(0, 0, n) \leftarrow n \geq 0$$
$$x + y = 3n \leftarrow p_0(x, y, n), x \geq n$$

Note that we only want to allow the user to modify the instantiation of predicates that occur in clauses that are not valid yet (here only p_0 qualifies). All other predicates are instantiated with the last assignment used by the solver. For this example, the user could come up with a variety of solutions. For example $2x = y \wedge x \leq n$ or $2y = x \wedge x \leq n$ which would be the invariant that we are actually looking for.

Each proposed solution can quickly be validated with an SMT solver by checking if all Horn clauses presented to the user are valid under the current assignment. If so, the solution is accepted and the user gets rewarded regardless of how useful the assignment is for solving the remaining clauses.

With the selected solutions, we can continue the Horn solving process. This will either lead to a solution or the solver will get stuck at a different position and create a new problem that can be crowd sourced.

Since we do not restrict which type of assignments a user can give to predicates (i.e., either only strengthen or only weaken the last solution), we cannot guarantee that this approach converges. Enforcing monotonicity can be enforced at low cost of an additional implication. However, as discussed in [29], forcing monotonicity is not necessarily the most practical way to find a solution, and from a crowdsourcing perspective, it might be confusing to disallow solutions.

We have implemented a prototype of the system and made it available online. At this point have not invested effort in usability and rely on the fact that users are experts in logic. Since the purpose of the paper is showing how Horn solving can be turned into a crowd sourcing problem, we focused on building the round-trip and leave a proper UI design and the development of an incentive system for future work.

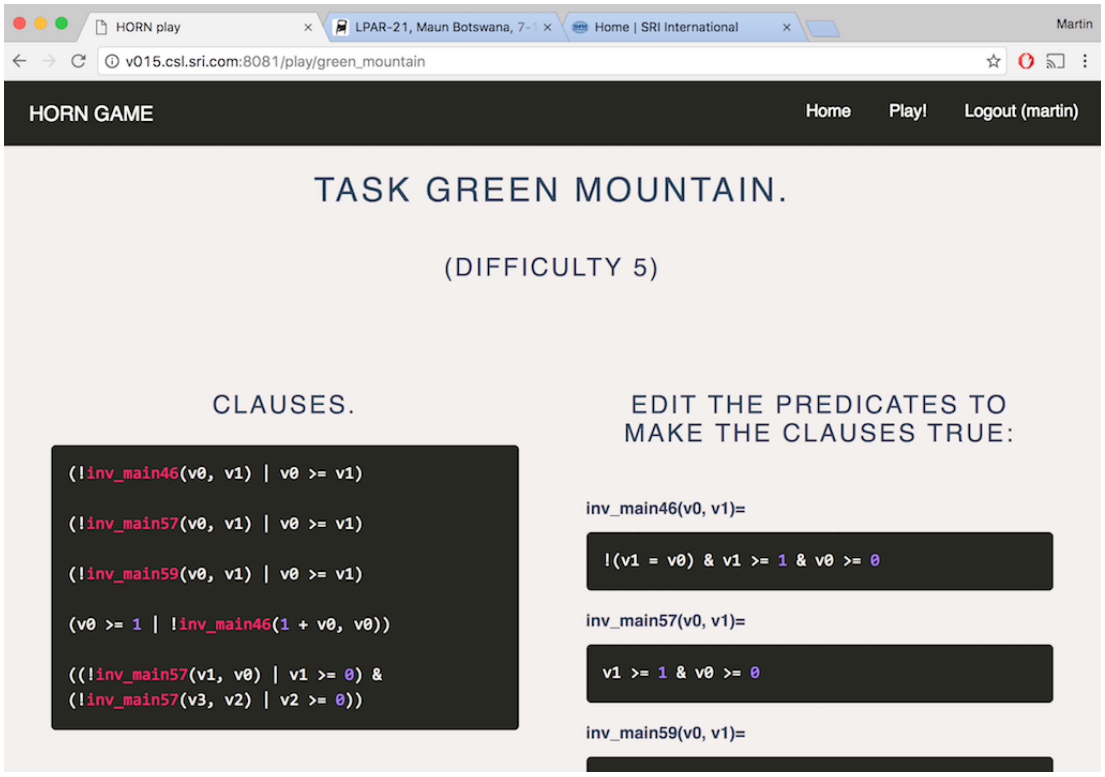
In Section 4 we discuss various design decisions in our implementation, how we perform book keeping, and how we create new problems.

3 From Horn clauses to Crowdsourcing

Many software verification problems can be encoded as CHCs. An overview of different approaches is given in [3] and several tools, such as SeaHorn [17], JayHorn [18], or the FLATA-C plugin for Frama-C [5], are able to generate verification conditions in the form of CHCs.

Throughout this paper, we use the following terminology: $\mathcal{H} := \mathsf{h}_1 \ldots \mathsf{h}_n$ is called a system of CHCs. A CHC h is a formula $H \leftarrow C \wedge B_1 \wedge \cdots \wedge B_n$ where B_i is an application of a relational symbol (or predicate) $p(exp_1, \ldots, exp_n)$ to terms exp_i in first-order logic, H is either also a predicate or *false*, and C is a constraint (e.g. $x < 5$). We say H is the head or the Horn clause and $(C \wedge B_1 \wedge \cdots \wedge B_n)$ is the body.

We use the shortcut $pred(\mathsf{h})$ to denote all predicate symbols p that occur in h and we use $\mathsf{h}|_{[p/\phi]}$ to denote a clause where all occurrences of predicate p have been instantiated by formula ϕ. An assignment Asn of formulas is a function that maps each predicate p to a formula



TASK GREEN MOUNTAIN.

(DIFFICULTY 5)

CLAUSES.

```
(!inv_main46(v0, v1) | v0 >= v1)

(!inv_main57(v0, v1) | v0 >= v1)

(!inv_main59(v0, v1) | v0 >= v1)

(v0 >= 1 | !inv_main46(1 + v0, v0))

((!inv_main57(v1, v0) | v1 >= 0) &
(!inv_main57(v3, v2) | v2 >= 0))
```

EDIT THE PREDICATES TO MAKE THE CLAUSES TRUE:

inv_main46(v0, v1)=

```
!(v1 = v0) & v1 >= 1 & v0 >= 0
```

inv_main57(v0, v1)=

```
v1 >= 1 & v0 >= 0
```

inv_main59(v0, v1)=

Figure 3: Screen shot of our web interface for providing assignments to predicates. The left shows the clauses where all predicates but the ones that can be changed have been instantiated, and the right shows the text box where the user can enter a new invariant.

$\phi = Asn(p)$. We say a CHC is solvable using symbolic models if there exists a Asn such that the formula $\mathsf{h}|_{[\forall p \in pred(horn), Asn(p)/p]}$ is valid.

Horn clauses solver implement different strategies to search for proper assignments. However, regardless of the implementation, they get stuck in similar ways, with a tentative assignment to the predicates that is not sufficient to make all clauses valid.

3.1 Crowdsourcing Symbolic Solutions

The Crowdsourcing Problem. *Given a system of CHCs with predicate symbols* $p_1, \ldots p_n$, *we say a* problem *is an assignment to each* p_i*, such that all clauses are satisfiable under this assignment but at least one clause is not valid. Solving this problem means modifying the assignments to* $p_1, \ldots p_n$ *until all Horn clauses become valid.*

Turning problems into tasks. Now that we have defined which problem we want to crowdsource, we can define how we split the problem into tasks that can be distributed and how solutions to the tasks can be checked efficiently. We propose to generate tasks from subsets of the CHCs, s.t. solutions are updates to the assignments of a subset of the predicates in the problem.

Each task has to contain at least one Horn clause that is not yet valid. Finding an update to the predicate assignments that makes this single clause valid should be an easy task a user (but the result may not be very useful). We can now tune the diffusivity of tasks by including additional clauses into the task. In our implementation, we include only clauses that share at least one predicate symbol with our initial clause. If the added clauses contain predicate symbols that are not part of our initial clause, we instantiate those to the assignments given by the problem. This ensures that the user can only modify the predicates that are relevant to the clause that we want to make valid.

Algorithm 1: Algorithm to generate crowdsourcing tasks from a given problem.

```
Input: Problem given as system of CHCs $\mathcal{H} := \mathsf{h}_1 \dots \mathsf{h}_n$ and assignment $Asn$, and a clause
       $\mathsf{h}_{task} \in \mathcal{H}$ that is satisfiable but not valid under $Asn$.
Output: A set of crowdsourcing tasks $T$, where each task is a triple $(\mathcal{H}, Asn)$ where $\mathcal{H}$ is the
        CHCs that have to be valid under this assignment, and $Asn$ is the current assignment
        under which all clauses in $\mathcal{H}$ are satisfiable but at least one is not valid.
begin
    $T \leftarrow \{\}$ ;
    $\mathcal{H}_{support} \leftarrow \{\mathsf{h} \in \mathcal{H} | \mathsf{h} \neq \mathsf{h}_{task} \wedge pred(\mathsf{h}) \cap pred(\mathsf{h}_{task}) \neq \emptyset\}$ ;
    for $\mathcal{H}_{sub} \in 2^{\mathcal{H}_{support}}$ do
        $\mathcal{H}_{task} \leftarrow \mathcal{H}_{sub}|_{[\forall p \notin pred(\mathsf{h}_{task}) : Asn(p)/p]}$;
        $\mathcal{H}_{task} \leftarrow \mathcal{H}_{task} \cup \{\mathsf{h}_{task}\}$;
        if All clauses in $\mathcal{H}_{task}$ are satisfiable then
            $T \leftarrow T \cup (pred(\mathsf{h}_{task}), \mathcal{H}_{task}, Asn)$;
    end for
    return $T$ ;
end
```

We describe our task generation in Algorithm 1. The algorithm takes a problem in the form of a system of CHCs $\mathcal{H}$ and an assignment Asn as input, s.t. all clauses in $\mathcal{H}$ are satisfiable under Asn and at least one is not valid. Further, the algorithm takes one $\mathsf{h}_{task} \in \mathcal{H}$ as input, s.t. h_{task} is satisfiable under Asn but not valid. If there are multiple clauses with that property in $\mathcal{H}$, we invoke the procedure separately for each of those clauses.

The algorithm now creates a set of tasks where the user has to find new assignments for the predicates $pred(\mathsf{h}_{task})$. To that end, it first creates a copy of $\mathcal{H}_{support}$ that contains the subset of clauses of $\mathcal{H}$ that share at least one predicate symbol with h_{task} (excluding h_{task}).

We iterate over all subsets of $\mathcal{H}_{support}$ to create a new task. First, we create a set of CHCs $\mathcal{H}_{task}$ that contains copies of the clauses in $\mathcal{H}_{support}$ where we instantiate all predicates in that are not $pred(\mathsf{h}_{task})$ to their current assignment in Asn. Then we add h_{task} to $\mathcal{H}_{task}$. That is, we ensure that $pred(\mathcal{H}_{task}) = pred(\mathsf{h}_{task})$ and that the user can only modify predicates that directly affect h_{task}.

To avoid generating unsolvable tasks, we check with an SMT solver that the conjunction of the partially instantiated clauses in $\mathcal{H}_{task}$ does not imply false.

If $\mathcal{H}_{task}$ is not unsolvable, we add a new task to our output. A task is a tuple consisting of the CHCs $\mathcal{H}_{task}$ together with the current assignment Asn. The user then has to provide an update to $Asn(p)$ for all $p \in pred(\mathcal{H})$.

Integrating crowdsourcing results. Users provide solutions to a crowdsourcing task $(\mathcal{H}, Asn)$ in the form of an updated assignments Asn'. To check if a solution should be accepted we can invoke an SMT solver to verify for each clause $\mathsf{h} \in \mathcal{H}$ if it is valid under Asn'. This also allows us to provide fine grained feedback if only some clauses are valid and other are not. This check is cheap since all predicates are instantiated and we can provide prompt feedback to the user.

If the user-provided solution is valid, it does not yet mean that the solution helps us to solve our original problem. We now have to seed our Horn solver with the Asn from the user. Since we still have the assignments from all predicates that were instantiated during generation of the task and the newly provided assignments, we can set the Horn solver to this assignment and let it perform its usual strategy from there. Either this solves our problem, or it gets stuck at a different position which then creates a new problem from which we can create new tasks.

Restarting the Horn solver is not done every time a user provides a solution since it is not clear how long the Horn solver will take. The user only obtains immediate feedback if the provided assignment makes all clauses valid.

This closes our crowdsourcing loop. We have demonstrated how Horn clause solving can be turned into a crowdsourcing problem, we have shown how the problem can be broken into tasks and how the difficulty of these tasks can be adjusted by limiting the number of clauses per tasks, and we have seen that user provided solutions can be checked with one SMT query per clause, and that the solution can seamlessly into the Horn solving process.

It is always a long stretch from theory to practice, so, in the following Section, we discuss some of the obstacles and epiphanies that came our way during the implementation.

4 Implementation

We give an overview of our crowdsourcing architecture along Figure 4. Our system gets verification problems formulated as CHCs from various sources, such as the software model checkers JayHorn and SeaHorn. The system is agnostic to where the clauses come from as long as they are in valid SMT2 format.

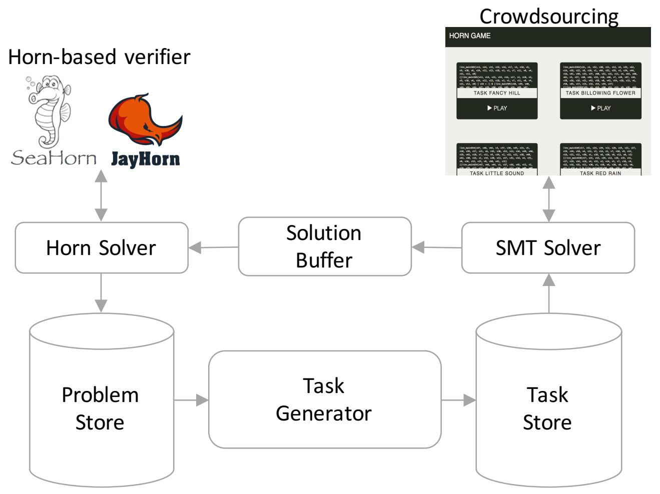


Figure 4: Overview of the architecture of our crowdsourcing infrastructure.

In the first step, our system checks to CHCs with two Horn solvers, Eldarica and Z3 with a configurable timeout. If the CHCs can be solved within the given time limit, we report the result immediately and no crowdsourcing is required. This can be seen as a portfolio solver. If any of the available solvers finds a solution, we return it, and it is easy to add additional solvers to the system, even if they do not support problem generation (because we only invoke problem generation if all solvers fail, in which case we can use the problem generation from Eldarica or Z3).

If the Horn solver fails to find a solution in the given time limit, it stops and returns an intermediate result as discussed in Sections **??** and **??**. The intermediate result then is added to the problem store as a json blob.

For each added problem, we invoke the task generator which, as discussed in Section 3.1, generates tasks by selecting a subset of clauses from a problem and instantiating the predicates that should not be modified by the user. Currently, our implementation generates all possible subset of clauses up to a size of six. In the evaluation, we discuss why our results suggest that this is not the most effective approach for crowdsourcing.

Each task gets checked with an SMT solver first the instantiated clauses are consistent. If this is not the case, the user would not be able to find a solution and we discard the task right away. All other tasks get stored in a database called task store.

The task store is connected to a Flask server to hosts the user interface. When users connect, they have to register first. We do not allow unregistered user to submit solutions to identify skills of users early. E.g., if a user tends to submit tautologies or trivial solutions, we do not want to waste time (and money on him), but if a user provides high quality solutions, we want to provide incentives for her to keep contributing.

Registered users can select from all available tasks. The tasks are sorted by *difficulty* (which is the number of clauses that need to be valid). If the user submits a candidate solution to a task, it is checked by the SMT solver. For each clause, we create an SMT query that checks if the clause is valid. Once all clauses are valid, the result get stored in a solution buffer.

A separate process fetches candidate solutions from the solution buffer, which contain the user assignments, and links to the task and the problem this task was created from. The process loads the SMT file associated with the problem and adds the assignments from the candidate solution as `initial-predicates` to the SMT file. Then it starts the Horn solver again. If the problem can now be solved, we mark the solution (to potentially award the player), and mark the problem and all tasks created by this problem for deletion. If the solver gets stuck under the new solution, it creates a new problem that gets added to the database.

5 Evaluation

We have implemented the system described in the previous section and are continuously collecting data to better understand its potential and limitations. The system is available online, and we encourage the reader to play with it and provide new solutions.[2]

We evaluate our system on a set of benchmark problems provided by Eldarica[3]. The benchmarks originate from different domains such as software assertion checking, protocol verification, and termination analysis. The SMT2 files were generated by different tools and we did not require any special formatting.

Experimental setup. We used a total of over 400 SMT files as input. For 219 of them, the Horn solvers could not find a solution within a 10 second time limit. From these, we created

[2] `http://www.horn-abduction.org`. Before playing, an account has to be created.
[3] `https://github.com/uuverifiers/eldarica`

the 219 initial problems for our crowdsourcing. All problems are unique and initially created a set of 219 tasks.

After several rounds of user submission, new tasks were created through refinement and, by the time of submitting this paper, are total number of 334 tasks have been generated.

To collect data from users, invited friends and colleges via email and social networks to try the website. We opened the website on January 20 to collect data and recorded the results after that.

5.1 Discussion

By the time of writing this paper, 58 users tried to solve problems on our website over and submitted a total 699 correct solutions. 37 of the 58 users registered but did not submit correct solutions.

Two players submitted a majority of the solutions. One player solver 221 problems and another player solver 133 problems. Since players are annonymous, we were not able to contact players. We investigated their submissions, and they submitted complex formulas within seconds are opening the problem, so we assume that they built a bot that uses a Horn solver to generate correct solutions. While this is against the idea of crowdsourcing, we are exited to see people trying to game our system and use it as a resful service.

So far, only the classical Horn problem *EvenOdd* could be solved by the crowd. Manual investigation of the remaining correct solutions revealed that the provided solutions indeed pointed into the right direction, but the subset of predicates that the user could influence was not sufficient to direct the Horn solver in the right direction. We did an in-depth analysis of several provided solutions and why they did not help to solve the underlying Horn problem. We identified four main problems in our design which we discuss below:

Encoding of Benchmarks. Encoding of clauses is crucial for humans to understand the clauses. For some of the SMT2 files, clauses used predicates with more than 50 parameters. While the clauses did not use all the parameters, the problems became almost unreadable. Few users even tried to solve these problems because the visual representation was just too discouraging. This could easily be avoided by introducing additional (local) predicates that use less variables. However, striking a good balance between a rewriting that is suitable for human consumption and that does not affect the Horn solver in a negative way is future work.

Task generation. For our experiments, we decided to only create tasks from the predicates that were assigned in the last unrolling of the Horn solver and to only present clauses to the user that contain these predicates (and instantiate all other predicates). For many tasks, this turned out to be a bad heuristic for two reasons: First, the predicate assigned in the last unrolling of the Horn solver might not be the one that needs fixing. Maybe an earlier assignment was bad and the user has no way of fixing the problem without touch this predicate. In particular, stopping the solver with a timeout gives us little control about where we stop. In the future, we need to mitigate this by either creating problems from multiple past unrollings, or to search for a particularly promising unrolling.

The second problem is that Horn clause solver tend to get stuck while searching for a fixpoint for a set of clauses with circular definition. If the subset of clauses that we use to generate the task does not contain all clauses of this circular definition, it hard to find, or even express that fixpoint. Several user complained that they could see that there is a recursive definition, but that one clause was missing, or that they could have solved it if they were allowed to edit other predicates. That is, in the future, we will experiment with different strategies to pick clauses during the task generation.

Trivial answers. One of the first things that users try is if they can get away with trivial answers like $false$ or trivial contradictions. When only presenting a subset of clauses, this is often the case. One way to avoid this problem is to check if the solution is satisfiable, but even then, users quickly find other ways of producing cheap answers, like repeating parts of the clauses. In the gamification setting, this can be tolerated but for payed crowdsourcing we need more mechanisms to ensure that players don't get payed for useless submissions. That is, we need checks that the solution is not trivial and that it is different from things the solver might have tried already. In the future, we will investigate how we can generate easily checkable restrictions for user-provided solutions.

Finding a cost effective solution that reduces the risk of getting (and checking) redundant or trivial answers, but that scales to large numbers of users is a challenging problem. A simple pair-wise comparison of submitted solutions quickly becomes impractical.

Moving in circles. Since we do not enforce that users submit answers that are strictly stronger or weaker than the last known assignment to a predicate, our search for a solution is not monotonic and thus we may get stuck in loops where we repeatedly generate equivalent tasks without making any progress. As for the trivial answers, this becomes an issue when users are paid for contributions. While this problem is not entirely avoidable due to the undecidability of the problem, we can ensure a certain progress by keeping track of previous solutions and tasks. We can check if one task is implied by a previous task before adding it to the database, and we can require the user to strengthen or weaken a known solution, instead of providing and arbitrary new result.

Summary. We have presented and implemented a crowdsourcing solution to assist Horn clauses solving. The approach is easy to implement and, due to the versatility of Horn clauses, can be applied to a multitude of verification problems. Building on top of Horn clauses eliminates many problems that previous crowdsourcing approaches faced, like controlling the difficulty of tasks and checking user-provided solutions efficiently.

While progress cannot be guaranteed, simple book keeping can help to avoid the creation of redundant tasks, and can disallow irrelevant answers.

Our initial assumption that we do not need to constrain the type of Horn clauses turned out to be problematic since there are limits to what humans are willing to solve. While our current system does not solve this problem, we have outlined some strategies in this discussion to mitigate the problem.

5.2 Threats to Validity

Before we can make any conclusions about the value of crowdsourcing for Horn solving and abduction we need to conduct a larger experiment. The current evaluation is only a feasibility test. In the future, we plan to perform a larger experiment where we control the skill level of users and distinguish the different type of Horn clause problems. Our assumption is that this approach shows its full potential for small but complicated Horn problems and with an audience that has a strong mathematical or puzzle solving background (rather than computer science). However, before we can run such a large study, more small experimentation like in this paper is needed to mature the work flow of the system and the presentation of problems.

6 Related Work

As discussed in the introduction, several projects investigated the idea of crowdsourcing verification tasks [7, 8, 12, 13, 21, 23, 30]. The main focus of these projects is to investigate to gamification potential and how to create a problem representation that is appealing to gamers. We are interested in the more general question of how verification problems can be translated to crowdsourcing problems. That is, what is a suitable methodology to split a verification problem into crowdsourcable tasks, how the solutions can be checked efficiently, and how solutions can be utilized efficiently. In this paper, we do not attempt to make the problem representation visually appealing but we acknowledge that a better presentation is key to get a larger user base.

Keeping a human in the loop is a common practice in verification. For example, Ivy [27], generates graphical representations of counterexamples from infinite-state system proofs which then need to be generalized by the user. Unlike our approach, their work requires a certain degree of expert knowledge about verification and, in particular, knowledge about the system under verification. Interactive verification environments such as PVS [25], Isabelle [24], or Coq [2] are also similar in that they seek guidance from the user only if automated reasoning gets stuck. However, these systems require expert knowledge of the task at hand and an idea of the bigger picture and are thus hard to crowdsource.

The survey in [22] names several applications of crowdsourcing in software engineering, such as software construction, documentation generation, or bug finding, and names some examples of commercially successful crowdsourcing in this domain. These approaches are very different from the task proposed in this paper, but their results and the success of platforms like Bountify, Upwork, and TopCoder suggest that there is a sufficiently large group of qualified workers available to scale our approach.

Another direction for seeking outside help in verification is the combination of formal verification and machine learning. Pioneered by the Daikon tool [11], and followed by a steady stream of research. These tools learn program properties, usually in the form of invariants. All these approaches need data points from different executions and apply different learning techniques, such as decision trees [20], ice-based learning [14], or deep learning [26], to infer an invariants or refinement types [31]. The motivation of these approaches is similar: get help from the outside when verification gets stuck. But the approach is different in that our approach tries to utilize the human which only makes sense if time is not an important factor. On the other hand, our approach does not need concrete states and is thus easier to implement.

7 Conclusion

We have presented a theoretical framework how crowdsourcing can assist Horn solvers. The central idea is that Horn solvers can get stuck when searching for a symbolic model because they fail to identify connections between variables that are not explicitly named in the clauses. This can be expressed as an abduction problem which is know to be hard to solve automatically. Humans can have an edge over machines in this task because they can draw from experience when adding missing information and their ability to abstract irrelevant information quickly.

We demonstrate that the Horn clause formalism nicely address many key problems in crowdsourcing of how to split problems into tasks, how to control the difficulty of tasks, how to check solutions to tasks, and how to utilize these solutions to solve a problem. All these problems have a straightforward solution in the Horn clause mechanism and we demonstrate that crowdsourcing seamlessly integrates with the Horn solving process and discussed a prototype implementation. We show that our prototype works and identify several engineering and presentation challenges that need to be addressed before the approach becomes useful in practice.

We do not want to overstate the applicability of crowdsourcing: Crowdsourcing is a very slow process; we have no influence on how long it takes until a smart user gets assigned to the problem that we are interested in. And crowdsourcing can only work if there is a solution in a logic that is decidable by the Horn solver. Hence, we emphasize that the presented approach only makes sense in domains where we need to verify large amounts of problems that are not time critical and for which we assume that they are solvable. However, for these cases, crowdsourcing can indeed achieve quasi-automation. And, depending on the user interface, the generated tasks may even have an educational aspect to teach users about Horn clauses, implications, and logical abduction.

References

[1] David P Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@ home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, 2002.

[2] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1.* PhD thesis, Inria, 1997.

[3] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In *Fields of Logic and Computation II*, pages 24–51. Springer, 2015.

[4] Nikolaj Bjørner, Kenneth L McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. *SMT@ IJCAR*, 20:3–11, 2012.

[5] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, 2012.

[6] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[7] Drew Dean, Sean Gaurino, Leonard Eusebi, Andrew Keplinger, Tim Pavlik, Ronald Watro, Aaron Cammarata, John Murray, Kelly McLaughlin, John Cheng, and Thomas Maddern. Lessons learned in game development for crowdsourced software formal verification. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, Washington, D.C., 2015. USENIX Association.

[8] Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Nathaniel Mote, Brian Walker, Seth Cooper, Timothy Pavlik, and Zoran Popović. Verification games: Making verification fun. In *FTfJP 2012: 14th Workshop on Formal Techniques for Java-like Programs*, pages 42–49, Beijing, China, June 12, 2012.

[9] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive invariant generation via abductive inference. In *ACM SIGPLAN Notices*, volume 48, pages 443–456. ACM, 2013.

[10] Christopher B Eiben, Justin B Siegel, Jacob B Bale, Seth Cooper, Firas Khatib, Betty W Shen, Barry L Stoddard, Zoran Popovic, and David Baker. Increased diels-alderase activity through backbone remodeling guided by foldit players. *Nature biotechnology*, 30(2):190–192, 2012.

[11] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*

[12] Daniel Fava, Dan Shapiro, Joseph Osborn, Martin Schäef, and E. James Whitehead, Jr. Crowdsourcing program preconditions via a classification game. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 1086–1096, New York, NY, USA, 2016. ACM.

[13] Daniel Fava, Julien Signoles, Matthieu Lemerre, Martin Schäf, and Ashish Tiwari. Gamifying program analysis. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 591–

605. Springer Berlin Heidelberg, 2015.

[14] Pranav Garg, Christof Löding, P Madhusudan, and Daniel Neider. Ice: A robust framework for learning invariants. In *International Conference on Computer Aided Verification*, pages 69–87. Springer, 2014.

[15] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12*, pages 405–416, 2012.

[16] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[17] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 343–361, 2015.

[18] Temesghen Kahsai, Phillip Rummer, Huascar Sanchez, and Martin Schaf. JayHorn: A framework for verifying java programs. In *Computer Aided Verification (CAV'16)*, 2016. To appear.

[19] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[20] Siddharth Krishna, Christian Puhrsch, and Thomas Wies. Learning invariants using decision trees. *arXiv preprint arXiv:1501.04725*, 2015.

[21] Heather Logas, Jim Whitehead, Michael Mateas, Richard Vallejos, Lauren Scott, Daniel G Shapiro, John Murray, Kate Compton, Joseph C Osborn, Orlando Salvatore, et al. Software verification games: Designing xylem, the code of plants. In *FDG*, 2014.

[22] Ke Mao, Licia Capra, Mark Harman, and Yue Jia. A survey of the use of crowdsourcing in software engineering. *Journal of Systems and Software*, (To appear), 2016.

[23] Kerry Moffitt, John Ostwald, Ron Watro, and Eric Church. Making hard fun in crowdsourced model checking: Balancing crowd engagement and efficiency to maximize output in proof by games. In *Proceedings of the Second International Workshop on CrowdSourcing in Software Engineering*, CSI-SE '15, pages 30–31, Piscataway, NJ, USA, 2015. IEEE Press.

[24] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[25] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[26] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 42–56, New York, NY, USA, 2016. ACM.

[27] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: Safety verification by interactive generalization. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 614–630, New York, NY, USA, 2016. ACM.

[28] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *Proceedings of the 25th International Conference on Computer Aided Verification*, CAV'13, pages 347–363, Berlin, Heidelberg, 2013. Springer-Verlag.

[29] Daniel Schwartz-Narbonne, Philipp Rümmer, Martin Schäf, Ashish Tiwari, and Thomas Wies. Non-monotonic program analysis. In *Proceedings 2nd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2015, San Francisco, CA.*, 2015.

[30] Ronald Watro, Kerry Moffitt, Talib Hussain, Daniel Wyschogrod, John Ostwald, Derrick Kong, Clint Bowers, Eric Church, Joshua Guttman, and Qinsi Wang. Ghost map: Proving software correctness using games. *SECURWARE 2014*, page 223, 2014.

[31] He Zhu, Aditya V Nori, and Suresh Jagannathan. Learning refinement types. In *ACM SIGPLAN Notices*, volume 50, pages 400–411. ACM, 2015.