Horn Clause Solvers for Program Verification

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan and Andrey Rybalchenko

Microsoft Research, Software Engineering Institute

Abstract. Automatic program verification and symbolic model checking tools interface with theorem proving technologies that check satisfiability of formulas. A theme pursued in the past years by the authors of this paper has been to encode symbolic model problems directly as Horn clauses and develop dedicated solvers for Horn clauses. Our solvers are called Duality, HSF, SeaHorn, and μZ and we have devoted considerable attention in recent papers to algorithms for solving Horn clauses. This paper complements these strides as we summarize main useful properties of Horn clauses, illustrate encodings of procedural program verification into Horn clauses and then highlight a number of useful simplification strategies at the level of Horn clauses. Solving Horn clauses amounts to establishing Existential positive Fixed-point Logic formulas, a perspective that was promoted by Blass and Gurevich.

1 Introduction

We make the overall claim that *Constrained Horn Clauses* provide a suitable basis for automatic program verification, that is, symbolic model checking. To substantiate this claim, this paper provides a self-contained, but narrowly selected, account for the use of Horn clauses in symbolic model checking. It is based on experiences the authors had while building tools for solving Horn clauses. At the practical level, we have been advocating the use of uniform formats, such as the SMT-LIB [6] standard as a format for representing and exchanging symbolic model checking problems as Horn clauses. The authors and many of our colleagues have developed several tools over the past years that solve Horn clauses in this format. We illustrate three approaches, taken from Duality, SeaHorn and HSF, for translating procedural programs into Horn clauses. At the conceptual level, Horn clause solving provides a uniform setting where we can discuss algorithms for symbolic model checking. This uniform setting allows us to consider integration of separate algorithms that operate as transformations of Horn clauses. We illustrate three transformations based on recent symbolic model checking literature and analyze them with respect to how they simplify the task of fully solving clauses. As a common feature, we show how solutions to the simplified clauses can be translated back to original clauses by means of Craig interpolation [22].

1.1 Program Logics and Horn Clauses

Blass and Gurevich [15] made the case that Existential positive Least Fixed-point Logic (E+LFP) provides a logical match for Hoare logic: Partial correctness of

simple procedural imperative programs correspond to satisfiability in E+LFP. We can take this result as a starting point for our focus on Horn clauses. As we show in Section 2.1, the negation of an E+LFP formula can be written as set of Horn clauses, such that the negation of an E+LFP formula is false if and only if the corresponding Horn clauses are satisfiable.

The connections between Constrained Horn Clauses and program logics originates with Floyd-Hoare logic [53, 29, 37]. Cook's [21] result on relative completeness with respect to Peano arithmetic established that Hoare's axioms were complete for safety properties relative to arithmetic. Clarke [20] established boundaries for relative completeness. Cook's result was refined by Blass and Gurevich.

In the world of constraint logic programming, CLP, expressing programs as Horn clauses and reasoning about Horn clauses has been pursued for several years, spearheaded by Joxan Jaffar and collaborators [41]. The uses of CLP for program analysis is extensive and we can only mention a few other uses of CLP for program analysis throughout the paper. Note that the more typical objective in constraint logic programming [2, 42] is to use logic as a declarative programming language. It relies on an execution engine that finds a set of *answers*, that is a set of substitutions that are solutions to a query. In an top-down evaluation engine, each such substitution is extracted from a refutation proof.

In the world of deductive databases [19], bottom-up evaluation of Datalog programs has, in addition to top-down, been explored extensively. Bottom-up evaluation infers consequences from facts and project the consequences that intersect with a query. Each such intersection corresponds to a refutation proof of a statement of the form "query is unreachable". Note that if the intersection is empty, then the smallest set of consequences closed under a Datalog program is a least model of the program and negated query.

Rybalchenko demonstrated how standard proof rules from program verification readily correspond to Horn clauses [32], and we have since been promoting constrained Horn clauses as a basis for program analysis [12].

1.2 Paper Outline

Figure 1 summarizes a use of Horn clauses in a verification workflow. Sections 3 and 4 detail translation of programs into clauses and simplifying transformations on clauses, respectively. Section 2 treat Horn clause basics. It is beyond the scope of this paper to go into depth of any of the contemporary methods for solving clauses, although this is central to the overall picture.

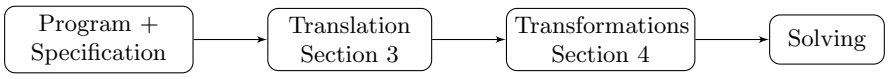

Fig. 1. Horn Clause verification flow

In more detail, in Section 2, we recall the main styles of Horn clauses used in recent literature and tools. We also outline contemporary methods for solving clauses that use strategies based on combinations of top-down and bottom-up search. As the main objective of solving Horn clauses is to show *satisfiability*, in contrast to showing that there is a derivation of the empty clause, we introduce a notion of models definable modulo an assertion language. We call these *symbolic models*. Many (but not all) tools for Horn clauses search for symbolic models that can be represented in a decidable assertion language. Note that symbolic models are simply synonymous to loop invariants, and [16] demonstrated that decidable assertion languages are insufficient for even a class of very simple programs. Section 3 compares some of the main approaches use for converting procedural programs into clauses. The approaches take different starting points on how they encode procedure calls and program assertions and we discuss how the resulting Horn clauses can be related. Section 4 summarizes three selected approaches for transforming Horn clauses. Section 4.1 recounts a query-answer transformation used by Gallagher and Kafle in recent work [30, 46]. In Section 4.2 we recall the well-known fold-unfold transformation and use this setting to recast K-induction [64] in the form of a Horn clause transformation. Section 4.3 discusses a recently proposed optimization for simplifying symbolic model checking problems [49]. We show how the simplification amounts to a rewriting strategy of Horn clauses. We examine each of the above transformation techniques under the lens of symbolic models, and address how they influence the existence and complexity of such models. The treatment reveals a common trait: the transformations we examine preserve symbolic models if the assertion language admits interpolation.

2 Horn Clause Basics

Let us first describe constrained Horn clauses and their variants. We take the overall perspective that constrained Horn clauses correspond to a fragment of first-order formulas modulo background theories.

We will assume that the *constraints* in constrained Horn Clauses are formulated in an assertion language that we refer to as $\mathcal{A}$. In the terminology of CLP, an assertion language is a constraint theory. In the terminology of SMT, an assertion language is a logic [6]. The terminology *assertion language* is borrowed from [52]. Typically, we let $\mathcal{A}$ be quantifier-free (integer) linear arithmetic. Other examples of $\mathcal{A}$ include quantifier-free bit-vector formulas and quantifier-free formulas over a combination of arrays, bit-vectors and linear arithmetic. Interpretations of formulas over $\mathcal{A}$ are defined by *theories*. For example, integer linear arithmetic can be defined by the signature $\langle Z, +, \leq \rangle$, where Z is an enumerable set of constants interpreted as the integers, $+$ is a binary function and $\leq$ is a binary predicate over integers interpreted as addition and the linear ordering on integers.

Schematic examples of constrained Horn clauses are

$$\forall x, y, z \;.\; q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow p(x, y)$$

and

$$\forall x, y, z \;.\; q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow \psi(z, x)$$

where p, q, r are predicate symbols of various arities applied to variables x, y, z and φ, ψ are formulas over an assertion language $\mathcal{A}$. More formally,

Definition 1 (CHC: Constrained Horn Clauses). *Constrained Horn clauses are constructed as follows:*

$$\begin{aligned}
\Pi &::= chc \wedge \Pi \mid \top \\
chc &::= \forall var \;.\; chc \mid body \rightarrow head \\
pred &::= upred \mid \varphi \\
head &::= pred \\
body &::= \top \mid pred \mid body \wedge body \mid \exists var \;.\; body \\
upred &::= \textit{an uninterpreted predicate applied to terms} \\
\varphi &::= \textit{a formula whose terms and predicates are interpreted over } \mathcal{A} \\
var &::= \textit{a variable}
\end{aligned}$$

We use P, Q, R as uninterpreted atomic predicates and B, C as bodies. A clause where the head is a formula φ is called a query *or a* goal clause*. Conversely we use the terminology* fact clause *for a clause whose head is an uninterpreted predicate and body is a formula φ.*

Note that constrained Horn clauses correspond to clauses that have at most one positive occurrence of an uninterpreted predicate. We use Π for a conjunction of constrained Horn clauses and *chc* to refer to a single constrained Horn clause.

Convention 1 *In the spirit of logic programming, we write Horn clauses as rules and keep quantification over variables implicit. Thus, we use the two representations interchangeably:*

$$\forall x, y, z \;.\; q(y) \wedge r(z) \wedge \varphi(x, y, z) \rightarrow p(x) \qquad as \qquad p(x) \leftarrow q(y), r(z), \varphi(x, y, z)$$

Example 1. Partial correctness for a property of the McCarthy 91 function can be encoded using the clauses

$$\begin{aligned}
mc(x, r) &\leftarrow x > 100, r = x - 10 \\
mc(x, r) &\leftarrow x \leq 100, y = x + 11, mc(y, z), mc(z, r) \\
r = 91 &\leftarrow mc(x, r), x \leq 101
\end{aligned}$$

The first two clauses encode McCarthy 91 as a constraint logic program. The last clause encodes the integrity constraint stipulating that whenever the McCarthy 91 function is passed an argument no greater than 101, then the result is 91.

Some formulas that are not directly Horn can be transformed into Horn clauses using a satisfiability preserving Tseitin transformation. For example, we can convert [1]

$$p(x) \leftarrow (q(y) \vee r(z)), \varphi(x, y, z) \tag{1}$$

into

$$s(y, z) \leftarrow q(y) \qquad s(y, z) \leftarrow r(z) \qquad p(x) \leftarrow s(y, z), \varphi(x, y, z) \tag{2}$$

by introducing an auxiliary predicate $s(y, z)$.

A wider set of formulas that admit an equi-satisfiable transformation to constrained Horn clauses is given where the body can be brought into negation normal form, NNF, and the head is a predicate or, recursively, a conjunction of clauses. When we later in Section 3 translate programs into clauses, we will see that NNF Horn clauses fit as a direct target language. So let us define the class of NNF Horn clauses as follows:

Definition 2 (NNF Horn).

$$\begin{aligned} \Pi &::= chc \wedge \Pi \mid \top \\ chc &::= \forall var\ .\ chc \mid body \rightarrow \Pi \mid head \\ head &::= pred \\ body &::= body \vee body \mid body \wedge body \mid pred \mid \exists var\ .\ body \end{aligned}$$

The previous example suggests there is an overhead associated with converting into constrained Horn clauses.

Proposition 1. *NNF Horn clauses with n sub-formulas and m variables can be converted into $O(n)$ new Horn clauses each using $O(m)$ variables.*

Thus, the size of the new formulas is $O(n \cdot m)$ when converting NNF Horn clauses into Horn clauses. The asymptotic overhead can be avoided by introducing a theory of tupling with projection and instead pass a single variable to intermediary formulas. For the formula (1), we would create the clauses:

$$s(u) \leftarrow q(\pi_1(u)) \qquad s(u) \leftarrow r(\pi_2(u)) \qquad p(x) \leftarrow s(\langle y, z \rangle), \varphi(x, y, z) \tag{3}$$

where π_1, π_2 take the first and second projection from a tuple variable u, and the notation $\langle x, y \rangle$ is used to create a tuple out of x and y.

Several assertion languages used in practice have *canonical* models. For example, arithmetic without division has a unique standard model. On the other

[1] Note that we don't need the clause $s(x, y) \rightarrow q(y) \vee r(z)$ to preserve satisfiability because the sub-formula that $s(x, y)$ summarizes is only used in negative scope.

hand, if we include division, then division by 0 is typically left under-specified and there is not a unique model, but many models, for formulas such as $x/0 > 0$.

Recall the notion of convexity [55], here adapted to Horn clauses. We will establish that Horn clauses and an extension called universal Horn clauses are convex. We show that a further extension, called existential Horn clauses, is not convex as an indication of the additional power offered by existential Horn clauses. Let Π be a set of Horn clauses, then Π is convex if for every pair of uninterpreted atomic predicates P, Q:

$$\Pi \models P \vee Q \quad \text{iff} \quad \Pi \models P \ \text{ or } \ \Pi \models Q$$

Proposition 2. *Suppose $\mathcal{A}$ has a canonical model $\mathcal{I}(\mathcal{A})$, then Horn clauses over $\mathcal{A}$, where each head is an uninterpreted predicate, are convex.*

The proposition is an easy consequence of

Proposition 3. *Constrained Horn clauses over assertion languages $\mathcal{A}$ that have canonical models have unique least models.*

This fact is a well known basis of Horn clauses [40, 67, 25]. It can be established by closure of models under intersection, or as we do here, by induction on derivations:

Proof. Let $\mathcal{I}(\mathcal{A})$ be the canonical model of $\mathcal{A}$. The initial model $\mathcal{I}$ of Π is defined inductively by taking $\mathcal{I}_0$ as $\emptyset$ and $\mathcal{I}_{i+1} := \{r(c) \mid (r(x) \leftarrow \mathit{body}(x)) \in \Pi, \mathcal{I}_i \models \mathit{body}(c),\ c \text{ is a constant in } \mathcal{I}(\mathcal{A})\}$. The initial model construction stabilizes at the first limit ordinal ω with an interpretation $\mathcal{I}_\omega$. This interpretation satisfies each clause in Π because suppose $(r(x) \leftarrow \mathit{body}(x)) \in \Pi$ and $\mathcal{I}_\omega \models \mathit{body}(c)$ for $c \in \mathcal{I}(\mathcal{A})$. Then, since the body has a finite set of predicates, for some ordinal $\alpha < \omega$ it is the case that $\mathcal{I}_\alpha \models \mathit{body}(c)$ as well, therefore $r(c)$ is added to $\mathcal{I}_{\alpha+1}$.

To see that Proposition 2 is a consequence of least unique models, consider a least unique model $\mathcal{I}$ of Horn clauses Π, then $\mathcal{I}$ implies either P or Q or both, so every extension of $\mathcal{I}$ implies the same atomic predicate.

While constrained Horn clauses suffice directly for Hoare logic, we applied two kinds of extensions for parametric program analysis and termination. We used universal Horn clauses to encode templates for verifying properties of array-based systems [14].

Definition 3 (UHC). *Universal Horn clauses extend Horn clauses by admitting universally quantifiers in bodies. Thus, the body of a universal Horn clause is given by:*

$$\mathit{body} ::= \top \mid \mathit{body} \wedge \mathit{body} \mid \mathit{pred} \mid \forall \mathit{var}\ .\ \mathit{body} \mid \exists \mathit{var}\ .\ \mathit{body}$$

Proposition 4. *Universal Horn clauses are convex.*

Proof. The proof is similar as constrained Horn clauses, but the construction of the initial model does not finish at ω, see for instance [15, 14]. Instead, we treat universal quantifiers in bodies as infinitary conjunctions over elements in the

domain of $\mathcal{I}(\mathcal{A})$ and as we follow the argument from Proposition 3, we add $r(c)$ to the least ordinal greater than the ordinals used to establish the predicates in the bodies.

Existential Horn clauses can be used for encoding reachability games [9].

Definition 4 (EHC). *Existential Horn clauses extend Horn clauses by admitting existential quantifications in the head:*

$$\mathit{head} ::= \exists \mathit{var} \ . \ \mathit{head} \mid \mathit{pred}$$

Game formalizations involve handling fixed-point formulas that alternate least and greatest fixed-points. This makes it quite difficult to express using formalisms, such as UHC, that are geared towards solving only least fixed-points. So, as we can expect, the class of EHC formulas is rather general:

Proposition 5. *EHC is expressively equivalent to general universally quantified formulas over $\mathcal{A}$.*

Proof. We provide a proof by example. The clause $\forall x, y \ . \ p(x, y) \vee q(x) \vee \neg r(y)$, can be encoded as three EHC clauses

$$(\exists z \in \{0, 1\} \ . \ s(x, y, z)) \leftarrow r(y) \qquad p(x, y) \leftarrow s(x, y, 0) \qquad q(x) \leftarrow s(x, y, 1)$$

We can also directly encode satisfiability of UHC using EHC by Skolemizing universal quantifiers in the body. The resulting Skolem functions can be converted into *Skolem relations* by creating relations with one additional argument for the return value of the function, and adding clauses that enforce that the relations encode total functions. For example, $p(x) \leftarrow \forall y \ . \ q(x, y)$ becomes $p(x) \leftarrow sk(x, y), q(x, y)$, and $(\exists y \ . \ sk(x, y)) \leftarrow q(x, y)$. Note that (by using standard polarity reasoning, similar to our Tseitin transformation of NNF clauses) clauses that enforce sk to be functional, e.g., $y = y' \leftarrow sk(x, y), sk(x, y')$ are redundant because sk is introduced for a negative sub-formula.

As an easy corollary of Proposition 5 we get

Corollary 1. *Existential Horn clauses are* not *convex.*

2.1 Existential Fixed-point Logic and Horn Clauses

Blass and Gurevich [15] identified Existential Positive Fixed-point Logic (E+LFP) as a match for Hoare Logic. They established a set of fundamental model theoretic and complexity theoretic results for E+LFP. Let us here briefly recall E+LFP and the main connection to Horn clauses. For our purposes we will assume that least fixed-point formulas are *flat*, that is, they use the fixed-point operator at the top-level without any nesting. It is not difficult to convert formulas with arbitrary nestings into flat formulas, or even convert formulas with multiple simultaneous definitions into a single recursive definition for that matter. Thus, using the notation from [15], a flat E+LFP formula Θ is of the form:

$$\Theta : \mathbf{LET} \bigwedge_i p_i(x) \leftarrow \delta_i(x) \ \mathbf{THEN} \ \varphi$$

where p_i, δ_i range over mutually defined predicates and neither δ_i nor φ contain any `LET` constructs. Furthermore, each occurrence of p_i in δ_j, respectively φ is positive, and δ_j and φ only contain existential quantifiers under an even number of negations. Since every occurrence of the uninterpreted predicate symbols is positive we can convert the *negation* of a flat E+LFP formula to NNF-Horn clauses as follows:

$$\Theta' : \bigwedge_i \forall x(\delta_i(x) \rightarrow p_i(x)) \wedge (\varphi \rightarrow \bot)$$

Theorem 1. *Let Θ be a flat closed E+LFP formula. Then Θ is equivalent to false if and only if the associated Horn clauses Θ' are satisfiable.*

Proof. We rely on the equivalence:

$$\neg(\mathbf{LET} \bigwedge_i p_i(x) \leftarrow \delta_i(x) \ \mathbf{THEN} \ \varphi) \equiv \exists \boldsymbol{p} \ . \ (\bigwedge_i \forall x \ . \ \delta_i(x) \rightarrow p_i(x)) \wedge \neg\varphi[\boldsymbol{p}]$$

where $\boldsymbol{p}$ is a vector of the predicate symbols p_i. Since all occurrences of p_i are negative, when some solution for $\boldsymbol{p}$ satisfies the fixed-point equations, and also satisfies $\neg\varphi[\boldsymbol{p}]$, then the *least* solution to the fixed-point equations also satisfies $\neg\varphi[\boldsymbol{p}]$.

Another way of establishing the correspondence is to invoke Theorem 5 from [15], which translates E+LFP formulas into $\forall_1^1$ formulas. The negation is an $\exists_1^1$ Horn formula.

Remark 1. The logic U+LFP, defined in [15], is similar to our UHC. The differences are mainly syntactic in that UHC allows alternating universal and existential quantifiers, but U+LFP does not.

2.2 Derivations and Interpretations

Horn clauses naturally encode the set of reachable states of sequential programs, so satisfiable Horn clauses are program properties that hold. In contrast, unsatisfiable Horn clauses correspond to violated program properties. As one would expect, it only requires a finite trace to show that a program property does *not* hold. The finite trace is justified by a sequence of resolution steps, and in particular for Horn clauses, it is sufficient to search for SLD [2] style proofs. We call these *top-down derivations*.

Definition 5 (Top-down derivations). *A top-down derivation starts with a* goal *clause of the form $\varphi \leftarrow B$. It selects a predicate $p(x) \in B$ and resolves it with a clause $p(x) \leftarrow B' \in \Pi$, producing the clause $\varphi \leftarrow B \setminus p(x), B'$, modulo renaming of variables in B and B'. The derivation concludes when there are no predicates in the goal, and the clause is false modulo $\mathcal{A}$.*

That is, top-down inferences maintain a goal clause with only negative predicates and resolve a negative predicate in the goal with a clause in Π. Top-down methods based on infinite descent or cyclic induction close sub-goals when they are implied by parent sub-goals. Top-down methods can also use interpolants or inductive generalization, in the style of the IC3 algorithm [17], to close sub-goals. In contrast to top-down derivations, bottom-up derivations start with clauses that have no predicates in the bodies:

Definition 6 (Bottom-up derivations). *A bottom-up derivation maintains a set of* fact *clauses of the form $p(x) \leftarrow \varphi$. It then applies hyper-resolution on clauses (head $\leftarrow B$) $\in \Pi$, resolving away all predicates in B using fact clauses. The clauses are inconsistent if it derives a contradictory fact clause (which has a formula from $\mathcal{A}$ in the head).*

Bottom-up derivations are useful when working with abstract domains that have join and widening operations. Join and widening are operations over an abstract domain (encoded as an assertion language $\mathcal{A}$) that take two formulas φ and φ' and create a consequence that is entailed by both.

For constrained Horn clauses we have

Proposition 6 (unsat is r.e.). *Let $\mathcal{A}$ be an assertion language where satisfiability is recursively enumerable. Then unsatisfiability for constrained Horn clauses over $\mathcal{A}$ is r.e.*

Proof. Recall the model construction from Proposition 3. Take the initial model of the subset of clauses that have uninterpreted predicates in the head. Checking membership in the initial model is r.e., because each member is justified at level $\mathcal{I}_i$ for some $i < \omega$. If the initial model also separates from $\bot$, then the clauses are satisfiable. So assuming the clauses are unsatisfiable there is a finite justification (corresponding to an SLD resolution derivation [2]), of $\bot$. The constraints from $\mathcal{A}$ along the SLD chain are satisfiable.

From the point of view of program analysis, refutation proof corresponds to a sequence of steps leading to a bad state, a bug. Program proving is much harder that finding bugs: satisfiability for Horn clauses is generally not r.e.

Definition 7 ($\mathcal{A}$-definable models). *Let $\mathcal{A}$ be an assertion language, an $\mathcal{A}$-definable model assigns to each predicate $p(x)$ a formula $\varphi(x)$ over the language of $\mathcal{A}$.*

Example 2. A linear arithmetic-definable model for the *mc* predicate in Example 1 is as follows:

$$mc(x, y) := y \geq 91 \wedge (y \leq 91 \vee y \leq x - 10)$$

We can verify that the symbolic model for *mc* satisfies the original three Horn clauses. For example, $x > 100 \wedge y = x - 10$ implies that $y > 100 - 10$, so $y \geq 91$ and $y \leq x - 10$. Thus, $mc(x, r) \leftarrow x > 10, r = x - 10$ is true.

Presburger arithmetic and additive real arithmetic are not expressive enough to define all models of recursive Horn clauses, for example one can define multiplication using Horn clauses and use this to define properties not expressible with addition alone [16, 63]. When working with assertion languages, such as Presburger arithmetic we are interested in more refined notions of completeness:

Definition 8 ($\mathcal{A}$-preservation). *A satisfiability preserving transformation of Horn clauses from Π to Π' is $\mathcal{A}$-preserving if Π has an $\mathcal{A}$-definable model if and only if Π' has an $\mathcal{A}$-definable model.*

We are also interested in algorithms that are complete relative to $\mathcal{A}$. That is, if there is an $\mathcal{A}$-definable model, they will find one. In [47] we identify a class of universal sentences in the Bernays Schoenfinkel class and an associated algorithm that is relatively complete for the fragment. In a different context Revesz identifies classes of vector addition systems that can be captured in Datalog [61]. In [11] we investigate completeness as a relative notion between search methods based on abstract interpretation and property directed reachability.

2.3 Loose Semantics and Horn Clauses

A formula φ is satisfiable modulo a background theory $\mathcal{T}$ means that there is an interpretation that satisfies the axioms of $\mathcal{T}$ and the formula φ (with free variables x). Thus, in Satisfiability Modulo Theories jargon, the queries are of the form

$$\exists f \;.\; \mathcal{A}x(f) \;\wedge\; \exists x \;.\; \varphi \tag{4}$$

where f are the functions defined for the theory $\mathcal{T}$ whose axioms are $\mathcal{A}x$. The second-order existential quantification over f is of course benign because the formula inside the quantifier is equi-satisfiable.

When the axioms have a canonical model, this condition is equivalent to

$$\forall f \;.\; \mathcal{A}x(f) \;\rightarrow\; \exists x \;.\; \varphi \tag{5}$$

In the context of Horn clause satisfiability, the format (5) captures the proper semantics. To see why, suppose *unk* is a global unknown array that is initialized by some procedure we can't model, and consider the following code snippet and let us determine whether it is safe.

$$\ell_0 : \textbf{if } (\mathit{unk}[x] > 0) \textbf{ goto } : \mathit{error}$$

In this example, the interpretation of the array *unknown* is not fully specified. So there could be an interpretation of *unknown* where the error path is not taken. For example, if ℓ_0 is reached under a context where $\mathit{unk}[x]$ is known to always be non-positive, the program is safe. Consider one possible way to translate this snippet into Horn clauses that we denote by $\mathit{Safe}(\ell_0, \mathit{unk})$:

$$\forall x \;.\; (\top \rightarrow \ell_0(x)) \wedge (\ell_0(x) \wedge \mathit{unk}[x] > 0 \rightarrow \bot).$$

These clauses are satisfiable. For example, we can interpret the uninterpreted predicates and functions as follows: $unk := const(0)$, $\ell_0(x) := \top$, where we use $const(0)$ for the array that constantly returns 0. This is probably not what we want. For all that we know, the program is not safe. Proper semantics is obtained by quantifying over all loose models. This amounts to checking satisfiability of:

$$\forall unk \exists \ell_0 \ . \ Safe(\ell_0, unk)$$

which is equi-satisfiable to:

$$\forall unk, x \ . \ ((\top \rightarrow \ell_0(unk, x)) \quad \wedge \quad (\ell_0(unk, x) \wedge unk[x] > 0 \rightarrow \bot)).$$

which is easily seen to be false by instantiating with $unk := const(1)$.

3 From Programs to Clauses

The are many different ways to transition from programs to clauses. This section surveys a few of approaches used in the literature and in tools. The conceptually simplest way to establish a link between checking a partial correctness property in a programming language and a formulation as Horn clauses is to formulate an operational semantics as an interpreter in a constraint logic program and then specialize the interpreter when given a program. This approach is used in the VeriMAP [24] tool an by [30]. The methods surveyed here bypass the interpreter and produce Horn clauses directly. Note that it is not just sequential programs that are amenable to an embedding into Horn clauses. One can for instance model a network of routers as Horn clauses [51].

3.1 State Machines

A state machine starts with an initial configuration of state variables $\boldsymbol{v}$ and transform these by a sequence of steps. When the initial states and steps are expressed as formulas $init(\boldsymbol{v})$ and $step(\boldsymbol{v}, \boldsymbol{v}')$, respectively, then we can check safety of a state machine relatively to a formula $safe(\boldsymbol{v})$ by finding an inductive invariant $inv(\boldsymbol{v})$ such that [52]:

$$inv(\boldsymbol{v}) \leftarrow init(\boldsymbol{v}) \quad inv(\boldsymbol{v}') \leftarrow inv(\boldsymbol{v}) \wedge step(\boldsymbol{v}, \boldsymbol{v}') \quad safe(\boldsymbol{v}) \leftarrow inv(\boldsymbol{v}) \quad (6)$$

3.2 Procedural Languages

Safety of programs with procedure calls can also be translated to Horn clauses. Let us here use a programming language substrate in the style of the Boogie [4] system:

$$\begin{aligned}
\mathit{program} &::= \mathit{decl}^* \\
\mathit{decl} &::= \textbf{def } p(x) \ \{ \ \textbf{local } \boldsymbol{v}; S \ \} \\
S &::= x := E \mid S_1; S_2 \mid \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2 \mid S_1 \square S_2 \\
&\quad \mid \textbf{havoc } x \mid \textbf{assert } E \mid \textbf{assume } E \\
&\quad \mid \textbf{while } E \textbf{ do } S \mid y := p(E) \mid \textbf{goto } \ell \mid \ell : S \\
E &::= \text{arithmetic logical expression}
\end{aligned}$$

In other words, a program is a set of procedure declarations. For simplicity of presentation, we restrict each procedure to a single argument x, local variables $\boldsymbol{v}$ and a single return variable *ret*. Most constructs are naturally found in standard procedural languages. The non-conventional **havoc**(x) command changes x to an arbitrary value, and the statement $S_1 \square S_2$ non-deterministically chooses run either S_1 or S_2. We use $\boldsymbol{w}$ for the set of all variables in the scope of a procedure. For brevity, we write procedure declarations as **def** $p(x)$ $\{S\}$ and leave the return and local variable declarations implicit. All methods generalize to procedures that modify global state and take and return multiple values, but we suppress handling this here. We assume there is a special procedure called *main*, for the main entry point of the program. Notice that assertions are included in the programming language.

Consider the program schema in Fig. 2. The behavior of procedure q is defined by the formula ψ, and other formulas $init, \varphi_1, \varphi_2$ are used for pre- and post-conditions.

```
def main(x) {
    assume init(x);
    z := p(x);
    y := p(z);
    assert φ1(y);
}

def p(x) {
    z := q(x);
    ret := q(z);
    assert φ2(ret);
}

def q(x) {
    assume ψ(x, ret);
}
```

Fig. 2. Sample program with procedure calls

Weakest Preconditions If we apply Boogie directly we obtain a translation from programs to Horn logic using a weakest liberal pre-condition calculus [26]:

$$
\begin{aligned}
\mathsf{ToHorn}(\mathit{program}) &:= \mathit{wlp}(\mathit{Main}(), \top) \wedge \bigwedge_{\mathit{decl} \in \mathit{program}} \mathsf{ToHorn}(\mathit{decl}) \\
\mathsf{ToHorn}(\textbf{def } p(x)\ \{S\}) &:= \mathit{wlp}\begin{pmatrix} \textbf{havoc } x_0; \textbf{assume } x_0 = x; \\ \textbf{assume } p_{pre}(x); S, \end{pmatrix} p(x_0, \mathit{ret}) \\
\mathit{wlp}(x := E, Q) &:= \textbf{let } x = E \textbf{ in } Q \\
\mathit{wlp}((\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2), Q) &:= \mathit{wlp}(((\textbf{assume } E; S_1) \square (\textbf{assume } \neg E; S_2)), Q) \\
\mathit{wlp}((S_1 \square S_2), Q) &:= \mathit{wlp}(S_1, Q) \wedge \mathit{wlp}(S_2, Q) \\
\mathit{wlp}(S_1; S_2, Q) &:= \mathit{wlp}(S_1, \mathit{wlp}(S_2, Q)) \\
\mathit{wlp}(\textbf{havoc } x, Q) &:= \forall x \,.\, Q \\
\mathit{wlp}(\textbf{assert } \varphi, Q) &:= \varphi \wedge Q \\
\mathit{wlp}(\textbf{assume } \varphi, Q) &:= \varphi \rightarrow Q \\
\mathit{wlp}((\textbf{while } E \textbf{ do } S), Q) &:= \mathit{inv}(\boldsymbol{w}) \wedge \\
&\qquad \forall \boldsymbol{w} \,.\, \begin{pmatrix} ((\mathit{inv}(\boldsymbol{w}) \wedge E) \ \rightarrow \ \mathit{wlp}(S, \mathit{inv}(\boldsymbol{w}))) \\ \wedge\ ((\mathit{inv}(\boldsymbol{w}) \wedge \neg E) \ \rightarrow \ Q) \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
wlp(y := p(E), Q) &:= p_{pre}(E) \wedge (\forall r \;.\; p(E, r) \;\rightarrow\; Q[r/y]) \\
wlp(\textbf{goto}\ \ell, Q) &:= \ell(\boldsymbol{w}) \wedge Q \\
wlp(\ell : S, Q) &:= wlp(S, Q) \wedge (\forall \boldsymbol{w} \;.\; \ell(\boldsymbol{w}) \rightarrow wlp(S, Q))
\end{aligned}
$$

The rule for $\Box$ duplicates the formula Q, and when applied directly can cause the resulting formula to be exponentially larger than the original program. Efficient handling of join-points has been the attention of a substantial amount of research around *large block* encodings [10] and optimized verification condition generation [28, 50, 5, 33]. The gist is to determine when to introduce auxiliary predicates for join-points to find a sweet spot between formula size and ease of solvability. Auxiliary predicates can be introduced as follows:

$$
wlp((S_1 \Box S_2), Q) := wlp(S_1, p(\boldsymbol{w})) \wedge wlp(S_2, p(\boldsymbol{w})) \wedge \forall \boldsymbol{w} \;.\; (p(\boldsymbol{w}) \rightarrow Q)
$$

Procedures can be encoded as clauses in the following way: A procedure $p(x)$ is summarized as a relation $p(x, ret)$, where x is the value passed into the procedure and the return value is ret.

Proposition 7. *Let prog be a program. The formula* $\mathsf{ToHorn}(prog)$ *is NNF Horn.*

Proof. By induction on the definition of *wlp*.

Example 3. When we apply ToHorn to the program in Fig. 2 we obtain a set of Horn clauses:

$$
\begin{aligned}
main(x) &\leftarrow \top \\
\varphi_1(y) &\leftarrow main(x), init(x), p(x, z), p(z, y) \\
p_{pre}(x) &\leftarrow main(x), init(x) \\
p_{pre}(z) &\leftarrow main(x), init(x), p(x, z) \\
p(x, y) \wedge \varphi_2(y) &\leftarrow p_{pre}(x), q(x, z), q(z, y) \\
q_{pre}(x) &\leftarrow p_{pre}(x) \\
q_{pre}(z) &\leftarrow p_{pre}(x), q(x, z) \\
q(x, y) &\leftarrow q_{pre}(x), \psi(x, y)
\end{aligned}
$$

Error flag propagation The SeaHorn verification system [34] uses a special parameter to track errors. It takes as starting point programs where asserts have been replaced by procedure calls to a designated error handler *error*. That is, **assert** φ statements are replaced by **if** $\neg\varphi$ **then** $error()$. Furthermore, it assumes that each procedure is described by a set of *control-flow edges*, i.e., statements of the form $\ell_{in} : S; \textbf{goto}\ \ell_{out}$, where S is restricted to a sequential composition of assignments, assumptions, and function calls.

To translate procedure declarations of the form **def** $p(x)$ { S }, SeaHorn uses procedure summaries of the form

$$
p(x, ret, e_i, e_o),
$$

where *ret* is the return value, and the flags e_i, e_o track the error status at entry and the error status at exit. If e_i is true, then the error status is transferred. Thus, for every procedure, we have the fact:

$$p(x, ret, \top, \top) \leftarrow \top \ .$$

In addition, for the error procedure, we have:

$$error(e_i, e_o) \leftarrow e_o \ .$$

We will use *wlp* to give meaning to basic statements here as well, using the duality of *wlp* and pre-image. To translate procedure calls that now take additional arguments we require to change the definition of *wlp* as follows:

$$wlp(y := p(E), Q) := \forall r, err \ . \ p(E, r, e_i, err) \rightarrow Q[r/y, err/e_i].$$

where *err* is a new global variable that tracks the value of the error flag.

Procedures are translated one control flow edge at a time. Each label ℓ is associated with a predicate $\ell(x_0, \boldsymbol{w}, e_o)$. Additionally, the entry of a procedure p is labeled by the predicate $p_{init}(x_0, \boldsymbol{w}, e_o)$ and the exit of a procedure is labeled by a predicate $p_{exit}(x_0, ret, e_o)$. An edge links its entry $\ell_{in}(x_0, \boldsymbol{w}, e_o)$ with its exit $\ell_{out}(x_0, \boldsymbol{w'}, e'_o)$, which is an entry point into successor edges. The rules associated with the edges are formulated as follows:

$$\begin{aligned}
p_{init}(x_0, \boldsymbol{w}, \bot) &\leftarrow x = x_0 && \text{where } x \text{ occurs in } \boldsymbol{w} \\
p_{exit}(x_0, ret, \top) &\leftarrow \ell(x_0, \boldsymbol{w}, \top) && \text{for each label } \ell, \text{ and } ret \text{ occurs in } \boldsymbol{w} \\
p(x, ret, \bot, \bot) &\leftarrow p_{exit}(x, ret, \bot) \\
p(x, ret, \bot, \top) &\leftarrow p_{exit}(x, ret, \top) \\
\ell_{out}(x_0, \boldsymbol{w'}, e_o) &\leftarrow \ell_{in}(x_0, \boldsymbol{w}, e_i) \wedge \neg e_i \wedge \neg wlp(S, \neg(e_i = e_o \wedge \boldsymbol{w} = \boldsymbol{w'}))
\end{aligned}$$

A program is safe if the clauses compiled from the program together with:

$$\bot \leftarrow Main_{exit}(x, ret, \top)$$

are satisfiable.

Example 4. When we create clauses directly from program in Fig. 2 we get the following set of clauses:

$$\begin{aligned}
\bot &\leftarrow main(\bot, \top) \\
main(e_i, e_o) &\leftarrow init(x), p(x, z, e_i, e'_o), p(y, z, e'_o, e''_o), \neg\varphi_1(y), error(e''_o, e_o) \\
main(e_i, e_o) &\leftarrow init(x), p(x, y, e_i, e'_o), p(y, z, e'_o, e_o), \varphi_1(y) \\
p(x, ret, e_i, e_o) &\leftarrow q(x, z, e_i, e'_o), q(z, ret, e'_o, e''_o), \neg\varphi_2(ret), error(e''_o, e_o) \\
p(x, ret, e_i, e_o) &\leftarrow q(x, z, e_i, e'_o), q(z, ret, e'_o, e_o), \varphi_2(ret) \\
q(x, ret, e_i, e_o) &\leftarrow \psi(x, ret), e_i = e_o \\
p(x, ret, \top, \top) &\leftarrow \top \\
q(x, ret, \top, \top) &\leftarrow \top \\
main(\top, \top) &\leftarrow \top \\
error(e_i, e_o) &\leftarrow e_o
\end{aligned}$$

Transition Summaries The HSF tool [32] uses summary predicates that capture relations between the program variables at initial locations of procedures and their values at a program locations within the same calling context. Transition summaries are useful for establishing termination properties. Their encoding captures the well-known RHS (Reps-Horwitz-Sagiv) algorithm [60, 3] that relies on top-down propagation with tabling (for use of tabling in logic programming, see for instance [68]). Thus, let $\boldsymbol{w}$ be the variables x, ret, local variables $\boldsymbol{v}$ and program location π for a procedure p. Then the translation into Horn clauses uses predicates of the form:

$$p(\boldsymbol{w}, \boldsymbol{w}').$$

To translate a procedure call $\ell : y := q(E); \ell'$ within a procedure p, create the clauses:

$$\begin{aligned}
p(\boldsymbol{w}_0, \boldsymbol{w}_4) &\leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), \mathit{call}(\boldsymbol{w}_1, \boldsymbol{w}_2), q(\boldsymbol{w}_2, \boldsymbol{w}_3), \mathit{return}(\boldsymbol{w}_1, \boldsymbol{w}_3, \boldsymbol{w}_4) \\
q(\boldsymbol{w}_2, \boldsymbol{w}_2) &\leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}_1), \mathit{call}(\boldsymbol{w}_1, \boldsymbol{w}_2) \\
\mathit{call}(\boldsymbol{w}, \boldsymbol{w}') &\leftarrow \pi = \ell, x' = E, \pi' = \ell_{q_{init}} \\
\mathit{return}(\boldsymbol{w}, \boldsymbol{w}', \boldsymbol{w}'') &\leftarrow \pi' = \ell_{q_{exit}}, \boldsymbol{w}'' = \boldsymbol{w}[ret'/y, \ell'/\pi]
\end{aligned}$$

The first clause establishes that a state $\boldsymbol{w}_4$ is reachable from initial state $\boldsymbol{w}_0$ if there is a state $\boldsymbol{w}_1$ that reaches a procedure call to q and following the return of q the state variables have been updated to $\boldsymbol{w}_4$. The second clause summarizes the starting points of procedure q. So, if p can start at state $\boldsymbol{w}_0$ For assertion statements $\ell : \textbf{assert}\ \varphi; \ell'$, produce the clauses:

$$\begin{aligned}
\varphi(\boldsymbol{w}) &\leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}), \pi = \ell \\
p(\boldsymbol{w}_0, \boldsymbol{w}[\ell'/\pi]) &\leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}), \pi = \ell, \varphi(\boldsymbol{w})
\end{aligned}$$

Other statements are broken into basic blocks similar to the error flag encoding. For each basic block $\ell : S; \ell'$ in procedure p create the clause:

$$p(\boldsymbol{w}_0, \boldsymbol{w}'') \leftarrow p(\boldsymbol{w}_0, \boldsymbol{w}), \pi_0 = \ell, \pi'' = \ell', \neg \mathit{wlp}(S, (\boldsymbol{w} \neq \boldsymbol{w}''))$$

Finally, add the following clause for the initial states:

$$\mathit{main}(\boldsymbol{w}, \boldsymbol{w}) \leftarrow \pi = \ell_{\mathit{main}_{init}}.$$

Note that transition summaries are essentially the same as what we get from ToHorn. The main difference is that one encoding uses program labels as state variables, the other uses predicates. Otherwise, one can extract the pre-condition for a procedure from the states that satisfy $p(\boldsymbol{w}, \boldsymbol{w})$, and similarly the post-condition as the states that satisfy $p(\boldsymbol{w}, \boldsymbol{w}') \wedge \pi' = \ell_{exit}$. Conversely, given solutions to p_{pre} and p, and the predicates summarizing intermediary locations within p one can define a summary predicate for p by introducing program locations.

3.3 Proof Rules

The translations from programs to Horn clauses can be used when the purpose is to to check assertions of sequential programs. This methodology, however is insufficient for dealing with concurrent programs with recursive procedures, and there are other scenarios where Horn clauses are a by-product of establishing program properties. The perspective laid out in [32] is that Horn clauses are really a way to write down search for intermediary assertions in proof rules as constraint satisfaction problems. For example, many proof rules for establishing termination, temporal properties, for refinement type checking, or for rely-guarantee reasoning can be encoded also as Horn clauses.

As an example, consider the rules (6) for establishing invariants of state machines. If we can establish that each reachable step is well-founded, we can also establish termination of the state machine. That is, we may ask to solve for the additional constraints:

$$round(v, v') \leftarrow inv(v) \wedge step(v, v'). \qquad wellFounded(round). \tag{7}$$

The well-foundedness constraint on *round* can be enforced by restricting the search space of solutions for the predicate to only well-founded relations.

Note that in general a proof rule may not necessarily be complete for establishing a class of properties. This means that the Horn clauses that are created as a side-effect of translating proof rules to clauses may be unsatisfiable while the original property still holds.

4 Solving Horn Clauses

A number of sophisticated methods have recently been developed for solving Horn clauses. These are described in depth in several papers, including [43, 32, 38, 27, 63, 54, 48, 24, 23, 11]. We will not attempt any detailed survey of these methods here, but just mention that most methods can be classified according to some main criteria first mentioned in Section 2.2:

1. Top-down derivations. In the spirit of SLD resolution, start with a goal and resolve the goals with clauses. Derivations are cut off by using cyclic induction or interpolants. If the methods for cutting off all derivation attempts, one can extract models from the failed derivation attempts. Examples of tools based on top-down derivation are [38, 54, 48].
2. Bottom-up derivations start with clauses that don't have uninterpreted predicates in the bodies. They then derive consequences until sufficiently strong consequences have been established to satisfy the clauses. Examples of tools based on bottom-up derivation are [32].
3. Transformations change the set of clauses in various ways that are neither top-down nor bottom-up directed.

We devote our attention in this section to treat a few clausal transformation techniques. Transformation techniques are often sufficiently strong to solve clauses directly, but they can also be used as pre-processing or in-processing techniques in other methods. As pre-processing techniques, they can significantly simplify Horn clauses generated from tools [8] and they can be used to bring clauses into a useful form that enables inferring useful consequences [46].

4.1 Magic Sets

The query-answer transformation [30, 46], a variant of the Magic-set transformation [68], takes a set of horn clauses Π and converts it into another set Π^{qa} such that bottom-up evaluation in Π^{qa} simulates top down evaluation of of Π. This can be an advantage in declarative data-bases as the bottom-up evaluation of the transformed program avoids filling intermediary tables with elements that are irrelevant to a given query. In the context of solving Horn clauses, the advantage of the transformation is that the transformation captures some of the calling context dependencies making bottom-up analysis more precise.

The transformation first replaces each clause of the form $\varphi \leftarrow B$ in Π by a clause $g \leftarrow B, \neg\varphi$, where g is a fresh uninterpreted *goal* predicate. It then adds the goal clauses $g^q \leftarrow \top$, $\bot \leftarrow g^a$ for each goal predicate g. We use the superscripts a and q in order to create two fresh symbols for each symbol. Finally, for $p(x) \leftarrow P_1, \ldots, P_n, \varphi$ in Π the transformation adds the following clauses in Π^{qa}:

- Answer clause: $p^a(x) \leftarrow p^q(x), P_1^a, \ldots, P_n^a, \varphi$
- Query clauses: $P_j^q \leftarrow p^q(x), P_1^a, \ldots, P_{j-1}^a, \varphi$ for $j = 1, \ldots, n$.

Where, by $P_1, \ldots, P_n$ are predicates $p_1, \ldots, p_n$ applied to their arguments. Given a set of clauses Π, we call the clauses that result from the transformation just described Π^{qa}.

A symbolic solution to the resulting set of clauses Π^{qa} can be converted into a symbolic solution for the original clause Π and conversely.

Proposition 8. *Given a symbolic solution φ^q, φ^a, ψ_1^q, $\psi_1^a, \ldots, \psi_n^q$, ψ_n^a, to the predicates $p, p_1, \ldots, p_n$, then $p(x) := \varphi^q \rightarrow \varphi^a$, $P_1 := \psi_1^q \rightarrow \psi_1^a$, $\ldots$, $P_n := \psi_n^q \rightarrow \psi_n^a$ solves $p(x) \leftarrow P_1, \ldots, P_n, \varphi$. Conversely, any solution to the original clauses can be converted into a solution of the Magic clauses by setting the query predicates to $\top$ and using the solution for the answer predicates.*

Note how the Magic set transformation essentially inserts pre-conditions into procedure calls very much in the same fashion that the ToHorn and the transition invariant translation incorporates pre-conditions to procedure calls.

Remark 2. Section 4.3 describes transformations that eliminate pre-conditions from procedure calls. In some way, the Magic set transformation acts inversely to eliminating pre-conditions.

4.2 Fold/unfold

The fold/unfold transformation [18, 65, 66] is also actively used in systems that check satisfiability of Horn clauses [57, 36] as well as in the partial evaluation literature [45].

The *unfold* transformation resolves each positive occurrence of a predicate with all negative occurrences. For example, it takes a system of the form

$$\begin{array}{l} q(y) \leftarrow B_1 \\ q(y) \leftarrow B_2 \\ p(x) \leftarrow q(y), C \end{array} \qquad \text{into} \qquad \begin{array}{l} p(x) \leftarrow B_1, C \\ p(x) \leftarrow B_2, C \end{array} \tag{8}$$

To define this transformation precisely, we will use the notation $\phi|_\iota$ to mean the sub-formula of ϕ at syntactic position ι and $\phi[\psi]_\iota$ to mean ϕ with ψ substituted at syntactic position ι. Now suppose we have two NNF clauses $C_1 = H_1 \leftarrow B_1$ and $C_2 = p(\boldsymbol{x}) \leftarrow B_2$ such that for some syntactic position ι in B_1, $B_1|_\iota = p(\boldsymbol{t})$. Assume (without loss of generality) that the variables occurring in C_1 and C_2 are disjoint. The *resolvent* of C_1 and C_2 at position ι is $H_1 \leftarrow B_1[B_2\sigma]_\iota$, where σ maps $\boldsymbol{x}$ to $\boldsymbol{t}_i$. We denote this $C_1\langle C_2\rangle_\iota$. The *unfolding* of C_2 in C_1 is $C_1\langle C_2\rangle_{\iota_1} \cdots \langle C_2\rangle_{\iota_k}$ where $\iota_1 \ldots \iota_K$ are the positions in B_1 of the form $p(\boldsymbol{t})$. That is, unfolding means simultaneously resolving all occurrences of p.

The *unfold transformation* on p replaces each clause C_1 with the set of clauses obtained by unfolding all the p-clauses in C_1. The unfold transformation is a very frequently used pre-processing rule and we will use it later on in Section 4.3. It simplifies the set of clauses but does not change the search space for symbolic models. As we will see in many cases, we can use the tool of *Craig interpolation* [22] to characterize model preservation.

Proposition 9. *The unfold transformation preserves $\mathcal{A}$-definable models if $\mathcal{A}$ admits interpolation.*

Proof. Take for instance a symbolic model that contains the definition $p(x) := \varphi$ and satisfies the clauses on the right of (8) together with other clauses. Assume that the symbolic model also contains definitions $r_1(x) := \psi_1, \ldots, r_m(x) := \psi_m$ corresponding to other uninterpreted predicate symbols in B_1, B_2, C and in other clauses. Then $((B_1 \vee B_2) \rightarrow (C \rightarrow p(x)))[\varphi/p, \psi_1/r_1, \ldots, \psi_m/r_m]$ is valid and we can assume the two sides of the implication only share the variable y. From our assumptions, there is an interpolant $q(y)$.

We can do a little better than this in the case where there is exactly one p-clause $C : p(\boldsymbol{x}) \leftarrow B$. We say the *reinforced resolvent* of C with respect to clause $H \leftarrow B'$ at position ι (under the same conditions as above) is $H \leftarrow B'[p(\boldsymbol{t}) \wedge B\sigma]_\iota$. Instead of *replacing* the predicate $p(\boldsymbol{t})$ with its definition, we *conjoin* it with the definition. This is valid when there is exactly one p-clause. In this case the original clauses and the reinforced clauses have the same initial models (which can be seen by unfolding once the corresponding recursive definition for p). Reinforced resolution induces a corresponding notion of reinforced unfolding.

The reinforced unfold transformation on p applies only if there is exactly one p-clause. It replaces each clause C with the clause obtained by reinforced unfolding the unique p-clause in C. As an example:

$$\begin{array}{l} p(y) \leftarrow B \\ q(x) \leftarrow p(y), \phi \end{array} \quad \text{unfolds into} \quad \begin{array}{l} p(y) \leftarrow B \\ q(x) \leftarrow p(y), B, \phi \end{array} \tag{9}$$

Proposition 10. *The reinforced unfold transformation preserves $\mathcal{A}$-definable models if $\mathcal{A}$ admits interpolation.*

Proof. Consider the example of (9), and suppose we have a solution $\mathcal{I}$ for the unfolded system (the right-hand side). Let $p'(y)$ be an interpolant for the valid implication $B\mathcal{I} \rightarrow (p(y) \wedge \phi \rightarrow q(x))\mathcal{I}$. Taking the conjunction of p' with $\mathcal{I}(p)$, we obtain a solution for the original (left-hand side) system. This construction can be generalized to any number of reinforced resolutions on p by using the conjunction of all the interpolants (but only under the assumption that there is just one p-clause).

The *fold* transformation takes a rule $q(x) \leftarrow B$ and replaces B everywhere in other rules by $q(x)$. For example it takes a system of the form:

$$\begin{array}{l} q(x) \leftarrow B \\ p(x) \leftarrow B, C \\ r(x) \leftarrow B, C' \end{array} \quad \text{into} \quad \begin{array}{l} q(x) \leftarrow B \\ p(x) \leftarrow q(x), C \\ r(x) \leftarrow q(x), C' \end{array} \tag{10}$$

To create opportunities for the *fold* transformation, rules for simplification and creating new definitions should also be used. For example, the rule $q(x) \leftarrow B$ is introduced for a fresh predicate q when there are multiple occurrences of B in the existing Horn clauses.

Remark 3. The fold/unfold transformations do not refer to goal, sub-goals or fact clauses. Thus, they can be applied to simplify and solve Horn clauses independent of top-down and bottom-up strategies.

K-induction and reinforced unfold K-induction [64] is a powerful technique to prove invariants. It exploits the fact that many invariants become inductive when they are checked across more than one step. To establish that an invariant *safe* is 2-inductive for a transition system with initial state *init* and transition *step* it suffices to show:

$$\begin{array}{r} \mathit{init}(\boldsymbol{v}) \rightarrow \mathit{safe}(\boldsymbol{v}) \\ \mathit{init}(\boldsymbol{v}) \wedge \mathit{step}(\boldsymbol{v}, \boldsymbol{v}') \rightarrow \mathit{safe}(\boldsymbol{v}') \\ \mathit{safe}(\boldsymbol{v}) \wedge \mathit{step}(\boldsymbol{v}, \boldsymbol{v}') \wedge \mathit{safe}(\boldsymbol{v}') \wedge \mathit{step}(\boldsymbol{v}', \boldsymbol{v}'') \rightarrow \mathit{safe}(\boldsymbol{v}'') \end{array} \tag{11}$$

Formally, 2-induction can be seen as simply applying the reinforced unfold transformation on *safe*. That is, in NNF we have:

$$\mathit{safe}(\boldsymbol{v}') \leftarrow \mathit{init}(\boldsymbol{v}') \vee (\mathit{safe}(\boldsymbol{v}) \wedge \mathit{step}(\boldsymbol{v}, \boldsymbol{v}'))$$

which unfolds to:

$$safe(\boldsymbol{v}'') \leftarrow init(\boldsymbol{v}'') \vee (safe(\boldsymbol{v}') \wedge (init(\boldsymbol{v}') \vee (safe(\boldsymbol{v}) \wedge step(\boldsymbol{v}, \boldsymbol{v}')) \wedge step(\boldsymbol{v}', \boldsymbol{v}'')))$$

which is equivalent to the clauses above. We can achieve K-induction for arbitrary K by simply unfolding the original definition of *safe* $K-1$ times in itself. Checking that any given predicate ϕ is K-inductive amounts to plugging it in for *safe* and checking validity. Interestingly, given a certificate π of K-induction of ϕ and feasible interpolation [58], the proof of Proposition 10 gives us a way to solve the original clause set. This gives us an ordinary safety invariant whose size is polynomial in π (though for propositional logic it may be exponential in the size of the original problem and ϕ).

4.3 A Program Transformation for Inlining Assertions

To improve the performance of software model checking tools Gurfinkel, Wei and Chechik [35] used a transformation called *mixed semantics* that eliminated call stacks from program locations with assertions. It is used also in Corral, as described by Lal and Qadeer [49], as a pre-processing technique that works with sequential and multi-threaded programs. The SeaHorn verification tool [34] uses this technique for transforming intermediary representations. In this way, the LLVM infrastructure can also leverage the transformed programs. The technique transforms a program into another program while preserving the set of assertions that are provable. We will here be giving a logical account for the transformation and recast it at the level of Horn clauses. We will use Horn clauses that are created from the ToHorn transformation and we will then use Horn clauses created from the error flag encoding. We show in both cases that call stacks around assertions can be eliminated, but the steps are different. They highlight a duality between the two translation techniques: Boogie inserts predicates to encode *safe pre-conditions* to procedures. SeaHorn generates predicates to encode *unsafe post-conditions* of procedures. Either transformation eliminates the safe pre-condition or the unsafe post-condition.

Optimizing ToHorn Recall the Horn clauses from Example 3 that were extracted from Fig. 2. The clauses are satisfiable if and only if:

$$\varphi_2(y) \leftarrow init(x), \psi(x, z), \psi(z, y)$$
$$\varphi_1(y) \leftarrow init(x), \psi(x, z_1), \psi(z_1, z), \psi(z, z_2), \psi(z_2, y)$$

is true. There are two main issues with direct inlining: (1) the result of inlining can cause an exponential blowup, (2) generally, when a program uses recursion and loops, finite inlining is impossible.

As a sweet spot one can inline stacks down to assertions in order to create easier constraint systems. The transformation proposed in [35, 49] converts the original program into the program in Fig. 3.

```
def main(x) {
    assume init(x);
    z := p(x) □ goto p_e;
    y := p(z) □ x := z; goto p_e;
    assert φ1(y);
    assume ⊥;
p_e:
    z := q(x) □ goto q_e;
    y := q(z) □ x := z; goto q_e;
    assert φ2(y);
    assume ⊥;
q_e:
    assume ψ(x, y);
    assume ⊥;
}
```

```
def p(x) {
    z := q(x);
    ret := q(z);
    assume φ2(y);
}

def q(x) {
    assume ψ(x, ret);
}
```

Fig. 3. Program with partially inlined procedures

It has the effect of replacing the original Horn clauses by the set

$$\begin{aligned}
\varphi_1(y) &\leftarrow \mathit{init}(x), p(x,z), p(z,y) \\
\varphi_2(z) \wedge p_{pre}(z) &\leftarrow \mathit{init}(x), q(x,z_1), q(z_1,z) \\
\varphi_2(y) &\leftarrow \mathit{init}(x), p(x,z), q(z,z_1), q(z_1,y) \\
p_{pre}(x) &\leftarrow \mathit{init}(x) \\
p(x,y) &\leftarrow p_{pre}(x), q(x,z), q(z,y), \varphi_2(y) \\
q(x,y) &\leftarrow \psi(x,y)
\end{aligned} \tag{12}$$

Part of this transformation corresponds to simple inlining of the calling contexts, but the transformation has another effect that is not justified by resolution alone: The formula $\varphi_2(y)$ is used as an assumption in the second to last rule. The transformation that adds φ_2 as an assumption is justified by the following proposition:

Proposition 11. *The following clauses are equivalent:*

$$\begin{array}{ll} \varphi \leftarrow B & \qquad \varphi \leftarrow B \\ P \leftarrow B & \qquad P \leftarrow B, \varphi \end{array}$$

We could in fact have baked in this transformation already when generating Horn clauses by pretending that every **assert** is followed by a matching **assume**, or by defining:

$$\mathit{wlp}(\textbf{assert}\ \varphi, Q) := \varphi \wedge (\varphi \rightarrow Q)$$

Furthermore, the clauses from our running example are equi-satisfiable to:

$$\varphi_1(y) \leftarrow \mathit{init}(x), p(x,z), p(z,y) \tag{13}$$

$$
\begin{aligned}
\varphi_2(z) &\leftarrow \mathit{init}(x), q(x, z_1), q(z_1, z)\\
\varphi_2(y) &\leftarrow \mathit{init}(x), p(x, z), q(z, z_1), q(z_1, y)\\
p(x, y) &\leftarrow q(x, z), q(z, y), \varphi_2(y)\\
q(x, y) &\leftarrow \psi(x, y)
\end{aligned}
$$

These clauses don't contain p_{pre}. The place where p_{pre} was used is in the rule that defines p. To justify this transformation let us refer to a general set of Horn clauses Π, and

- Let $\mathcal{P} : C_1, C_2, \ldots$ be the clauses where P occurs negatively at least once.
- Let $\mathcal{R} : Q \leftarrow D_1, Q \leftarrow D_2, \ldots$ be the clauses where Q occurs positively and assume Q does not occur negatively in these clauses.

Proposition 12. *Let $P \leftarrow Q \wedge B$ be a clause in Π. Then Π is equivalent to $\{P \leftarrow B\} \cup \Pi$ if the following condition holds: For every clause $C \in \mathcal{P}$ let C' be the result of resolving all occurrences of P with $P \leftarrow Q \wedge B$, then there exists a sequence of resolvents for Q from $\mathcal{R}$, such that each resolvent subsumes C'.*

The intuition is of course that each pre-condition can be discharged by considering the calling context. We skip the tedious proof and instead give an example tracing how the proposition applies.

Example 5. Consider the clause $q(x, y) \leftarrow q_{pre}(x), \psi(x, y)$ from (3). We wish to show that $q_{pre}(x)$ can be removed from the premise. Thus, take for example the clause $q_{pre}(z) \leftarrow p_{pre}(x), q(x, z)$ where q occurs negatively. Then resolving with q produces C': $q_{pre}(z) \leftarrow p_{pre}(x), q_{pre}(x), \psi(x, y)$. The pre-condition is removed by resolving with $q_{pre}(x) \leftarrow p_{pre}(x)$, producing the subsuming clause $q_{pre}(z) \leftarrow p_{pre}(x), p_{pre}(x), \psi(x, y)$. A somewhat more involved example is the clause $p(x, y) \leftarrow p_{pre}(x), q(x, z), q(z, y)$. We will have to resolve against q in both positions. For the first resolvent, we can eliminate q_{pre} as we did before. Resolving against the second occurrence of q produces

$$p(x, y) \leftarrow p_{pre}(x), q(x, z), q_{pre}(z), \psi(z, y).$$

This time resolve with the clause $q_{pre}(z) \leftarrow p_{pre}(x), q(x, z)$ producing

$$p(x, y) \leftarrow p_{pre}(x), q(x, z), q(x', z), p_{pre}(x'), \psi(z, y),$$

which is equivalent to $p(x, y) \leftarrow p_{pre}(x), q(x, z), \psi(z, y)$.

The resulting Horn clauses are *easier* to solve: the burden to solve for p_{pre} has been removed, and the clauses that constrain P have been weakened with an additional assumption. However, similar to other transformations, we claim we can retrieve a solution for p_{pre} if $\mathcal{A}$ admits interpolation.

Error flag specialization We can arrive to the same result using *specialization* of the Horn clauses generated from Section 3.2 followed by inlining. The specialization step is to create fresh copies of clauses by grounding the values of the Booleans e_i and e_o.

Consider the clauses from Example 4. We *specialize* the clauses with respect to e_i, e_o by instantiating the clauses according to the four combinations of the e_i, e_o arguments. This reduction could potentially cause an exponential increase in number of clauses, but we can do much better: neither $p(x, y, \top, \bot)$ nor $q(x, y, \top, \bot)$ are derivable. This reduces the number of instantiations significantly from exponential to at most a linear overhead in the size of the largest clause. To reduce clutter, let $p_{fail}(x, y)$ be shorthand for $p(x, y, \bot, \top)$ and $p_{ok}(x, y)$ be shorthand for $p(x, y, \bot, \bot)$.

$$\begin{aligned}
\bot &\leftarrow main_{fail} \\
main_{fail} &\leftarrow init(x), p_{fail}(x, y) \\
main_{fail} &\leftarrow init(x), p_{ok}(x, y), p_{fail}(y, z) \\
main_{fail} &\leftarrow init(x), p_{ok}(x, y), p_{ok}(y, z), \neg\varphi_1(y) \\
p_{fail}(x, ret) &\leftarrow q_{fail}(x, z) \\
p_{fail}(x, ret) &\leftarrow q_{ok}(x, z), q_{fail}(z, ret) \\
p_{fail}(x, ret) &\leftarrow q_{ok}(x, z), q_{ok}(z, ret), \neg\varphi_2(ret) \\
p_{ok}(x, ret) &\leftarrow q_{ok}(x, z), q_{ok}(z, ret), \varphi_2(ret) \\
q_{ok}(x, ret) &\leftarrow \psi(x, ret)
\end{aligned} \tag{14}$$

In the end we get by unfolding the post-conditions for failure $main_{fail}$, p_{fail} and q_{fail}:

$$\begin{aligned}
\bot &\leftarrow init(x), q_{ok}(x, z), q_{ok}(z, y), \neg\varphi_2(y) \\
\bot &\leftarrow init(x), p_{ok}(x, y), q_{ok}(y, u), q_{ok}(u, z), \neg\varphi_2(z) \\
\bot &\leftarrow init(x), p_{ok}(x, y), p_{ok}(y, z), \neg\varphi_1(y) \\
p_{ok}(x, ret) &\leftarrow q_{ok}(x, z), q_{ok}(z, ret), \varphi_2(ret) \\
q_{ok}(x, ret) &\leftarrow \psi(x, ret)
\end{aligned} \tag{15}$$

which are semantically the same clauses as (13).

5 Conclusions and Continuations

We have described a framework for checking properties of programs by checking satisfiability of (Horn) clauses. We described main approaches for mapping sequential programs into Horn clauses and some main techniques for transforming Horn clauses. We demonstrated how many concepts developed in symbolic model checking can be phrased in terms of Horn clause solving. There are many extensions we did not describe here, and some are the focus of active research. Let us briefly mention a few areas here.

Games Winning strategies in infinite games use alternations between least and greatest fixed-points. Horn clauses are insufficient and instead [9] encodes games using EHC, which by Proposition 5 amounts to solving general universally quantified formulas.

Theories We left the assertion language $\mathcal{A}$ mostly unspecified. Current Horn clause solvers are mainly tuned for real and linear integer arithmetic and Boolean domains, but several other domains are highly desirable, including strings, bit-vectors, arrays, algebraic data-types, theories with quantifiers (EPR, the Bernays Schoenfinkel class). In general $\mathcal{A}$ can be defined over a set of templates or syntactically as formulas over a grammar for a limited language. For example, the sub-language of arithmetic where each inequality has two variables with coefficients ± 1 is amenable to specialized solving. Finally, one can also treat separation logic as a theory [56].

Consequences and abstraction interpretation in CLP While the strongest set of consequences from a set of Horn clauses is a least fixed-point over $\mathcal{A}$, one can use abstract domains to over-approximate the set of consequences. Thus, given a set of Horn clauses Π over assertion language $\mathcal{A}$ compute the strongest consequences over assertion language $\mathcal{A}' \subseteq \mathcal{A}$.

Classification There are several special cases of Horn clauses that can be solved using dedicated algorithms [63]. An example of "easier" clauses is *linear Horn clauses* that only contain at most one uninterpreted predicate in the bodies. Naturally, recursion-free Horn clauses can be solved whenever $\mathcal{A}$ is decidable. Horn clauses obtained from QBF problems with large blocks of quantified variables are solved more efficiently if one realizes that clauses can be rewritten corresponding to re-ordering variables.

Higher-order programs The interpreter approach for assigning meanings to programs can be extended to closures in a straight-forward way. Closures encode function pointers and state and they can be encoded when $\mathcal{A}$ supports algebraic data-types [13]. This allows establishing properties of functional programs where all closures are defined within the program. The more general setting was given a custom proof system in [31], and modern approaches to proving properties of higher-order rewriting systems use a finite state abstraction as higher-order Boolean programs [59]. A different approach extracts Horn clauses from refinement based type systems for higher-order programs [62, 44].

Beyond $\mathcal{A}$-definable satisfiability Our emphasis on $\mathcal{A}$-definable models is partially biased based on the methods developed by the authors, but note that methods based on superposition, infinite descent and fold/unfold can establish satisfiability of Horn clauses without producing a $\mathcal{A}$-definable model. Some other clausal transformation techniques we have not described are based on accelerating transitive relations [27, 39, 1].

Aggregates and Optimality Suppose we would like to say that a program has at most a $2 \cdot n$ reachable states for a parameter n. We can capture and solve such constraints by introducing cardinality operators that summarize the number of reachable states. Note that upper bounds constraints on cardinalities preserve least fixed-points: If there is a solution not exceeding a bound, then any conjunction of solutions also will not exceed a bound. Lower-bound constraints, on the other hand, are more subtle to capture. Rybalchenko et al. use a symbolic version of Barvinok's algorithm [7] to solve cardinality constraints. Instead of proving bounds, we may also be interested in finding solutions that optimize objective functions.

We would like to thank Dejan Jovanovich and two peer reviewers for extensive feedback on an earlier version of the manuscript.

References

1. Francesco Alberti, Silvio Ghilardi, and Natasha Sharygina. Booster: An acceleration-based verification framework for array programs. In *ATVA*, pages 18–23, 2014.
2. Krzysztof R. Apt. Logic programming. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 493–574. Elsevier, 1990.
3. Thomas Ball and Sriram K. Rajamani. Bebop: a path-sensitive interprocedural dataflow engine. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01, Snowbird, Utah, USA, June 18-19, 2001*, pages 97–103, 2001.
4. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO*, pages 364–387, 2005.
5. Michael Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, pages 82–87, 2005.
6. Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.
7. Alexander I. Barvinok. A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed. In *34th Annual Symposium on Foundations of Computer Science, Palo Alto, California, USA, 3-5 November 1993*, pages 566–572, 1993.
8. Josh Berdine, Nikolaj Bjørner, Samin Ishtiaq, Jael E. Kriener, and Christoph M. Wintersteiger. Resourceful reachability as HORN-LA. In *Logic for Programming, Artificial Intelligence, and Reasoning - LPAR*, pages 137–146, 2013.
9. Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, pages 221–234, 2014.
10. Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *FM-CAD*, pages 25–32, 2009.
11. Nikolaj Bjørner and Arie Gurfinkel. Property directed polyhedral abstraction. In *Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings*, pages 263–281, 2015.
12. Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program Verification as Satisfiability Modulo Theories. In *SMT at IJCAR*, pages 3–11, 2012.
13. Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Higher-order program verification as satisfiability modulo theories with algebraic data-types. *CoRR*, abs/1306.5264, 2013.
14. Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. On Solving Universally Quantified Horn Clauses. In *SAS*, pages 105–125, 2013.
15. A. Blass and Y. Gurevich. Existential fixed-point logic. In *Computation Theory and Logic*, pages 20–36, 1987.
16. Andreas Blass and Yuri Gurevich. Inadequacy of computable loop invariants. *ACM Trans. Comput. Log.*, 2(1):1–11, 2001.
17. Aaron R. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, pages 70–87, 2011.

18. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *JACM*, 24, 1977.
19. Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Springer, 1990.
20. Edmund M. Clarke. Programming Language Constructs for Which It Is Impossible To Obtain Good Hoare Axiom Systems. *J. ACM*, 26(1):129–147, 1979.
21. Stephen A. Cook. Soundness and completeness of an axiom system for program verif. *SIAM J. Comput.*, 7(1):70–90, 1978.
22. William Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. *J. Symb. Log.*, 22(3):269–285, 1957.
23. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175, 2014.
24. Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Verimap: A tool for verifying programs through transformations. In *TACAS*, pages 568–574, 2014.
25. Pilar Dellunde and Ramon Jansana. Some Characterization Theorems for Infinitary Universal Horn Logic Without Equality. *J. Symb. Log.*, 61(4):1242–1260, 1996.
26. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, 1976.
27. Arnaud Fietzke and Christoph Weidenbach. Superposition as a decision procedure for timed automata. *Mathematics in Computer Science*, 6(4):409–425, 2012.
28. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
29. R.W. Floyd. Assigning meaning to programs. In *Proceedings Symposium on Applied Mathematics*, volume 19, pages 19–32. American Math. Soc., 1967.
30. John P. Gallagher and Bishoksan Kafle. Analysis and Transformation Tools for Constrained Horn Clause Verification. *CoRR*, abs/1405.3883, 2014.
31. Steven M. German, Edmund M. Clarke, and Joseph Y. Halpern. Reasoning about procedures as parameters in the language L4. *Inf. Comput.*, 83(3):265–359, 1989.
32. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, 2012.
33. Arie Gurfinkel, Sagar Chaki, and Samir Sapra. Efficient predicate abstraction of program summaries. In *NASA Formal Methods - Third International Symposium, NFM*, pages 131–145, 2011.
34. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn Verification Framework. In *Submitted to CAV*, 2015.
35. Arie Gurfinkel, Ou Wei, and Marsha Chechik. Model checking recursive programs with exact predicate abstraction. In *Automated Technology for Verification and Analysis, 6th International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings*, pages 95–110, 2008.
36. Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro Lopez-Garcia, Edison Mera, José F. Morales, and Germán Puebla. An overview of ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.
37. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
38. Krystof Hoder and Nikolaj Bjørner. Generalized Property Directed Reachability. In *SAT*, pages 157–171, 2012.
39. Hossein Hojjat, Radu Iosif, Filip Konecný, Viktor Kuncak, and Philipp Rümmer. Accelerating interpolants. In *ATVA*, pages 187–202, 2012.

40. Alfred Horn. On Sentences Which are True of Direct Unions of Algebras. *J. Symb. Log.*, 16(1):14–21, 1951.
41. Joxan Jaffar. A CLP approach to modelling systems. In *ICFEM*, page 14, 2004.
42. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.
43. Joxan Jaffar, Andrew E. Santosa, and Razvan Voicu. An interpolation method for CLP traversal. In *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*, pages 454–469, 2009.
44. Ranjit Jhala, Rupak Majumdar, and Andrey Rybalchenko. HMC: verifying functional programs using abstract interpreters. In *CAV*, pages 470–485, 2011.
45. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Prentice Hall international series in computer science. Prentice Hall, 1993.
46. Bishoksan Kafle and John P. Gallagher. Constraint Specialisation in Horn Clause Verification. In *PEPM*, pages 85–90, 2015.
47. Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence, 2015.
48. Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-Based Model Checking for Recursive Programs. In *CAV*, pages 17–34, 2014.
49. Akash Lal and Shaz Qadeer. A program transformation for faster goal-directed search. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 147–154, 2014.
50. K. Rustan M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6):281–288, 2005.
51. Nuno P. Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *NSDI*, May 2015.
52. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.
53. John McCarthy. Towards a mathematical science of computation. In *IFIP Congress*, pages 21–28, 1962.
54. Kenneth L. McMillan. Lazy annotation revisited. In *CAV*, pages 243–259, 2014.
55. D. C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.
56. Juan Antonio Navarro Pérez and Andrey Rybalchenko. Separation logic modulo theories. In *Programming Languages and Systems - 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, pages 90–106, 2013.
57. A. Pettorossi and M. Proietti. Synthesis and transformation of logic programs using unfold/fold proofs. Technical Report 457, Universitá di Roma Tor Vergata, 1997.
58. P. Pudl'ak. Lower bounds for resolution and cutting planes proofs and monotone computations. *J. of Symbolic Logic*, 62(3):981–998, 1995.
59. Steven J. Ramsay, Robin P. Neatherway, and C.-H. Luke Ong. A type-directed abstraction refinement approach to higher-order model checking. In *POPL*, pages 61–72, 2014.
60. Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.
61. Peter Z. Revesz. Safe datalog queries with linear constraints. In *CP98, NUMBER 1520 IN LNCS*, pages 355–369. Springer, 1998.

62. Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *PLDI*, pages 159–169, 2008.
63. Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive Interpolants for Horn-Clause Verification. In *CAV*, pages 347–363, 2013.
64. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *FMCAD*, pages 108–125, 2000.
65. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Conference on Logic Programming*, 1984.
66. V. F. Turchin. The concept of a supercompiler. *ACM TOPLAS*, 8(3), 1986.
67. Maarten H. van Emden and Robert A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM*, 23(4):733–742, 1976.
68. David Scott Warren. Memoing for logic programs. *Commun. ACM*, 35(3):93–111, 1992.