Analysis and Transformation Tools for Constrained Horn Clause Verification∗

John P. Gallagher
Roskilde University, Denmark and IMDEA Software Institute, Madrid, Spain
(*e-mail:* `jpg@ruc.dk`)

Bishoksan Kafle
Roskilde University, Denmark
(*e-mail:* `kafle@ruc.dk`)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Several techniques and tools have been developed for verification of properties expressed as Horn clauses with constraints over a background theory (CHC). Current CHC verification tools implement intricate algorithms and are limited to certain subclasses of CHC problems. Our aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve challenging CHC verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general CHC verification tools based on a library of components.

KEYWORDS: Constraint Logic Program, Constrained Horn Clause, Abstract Interpretation, Software Verification.

1 Introduction

CHCs provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). As a result it has been used as a target language for software verification. Recently there is a growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC.

Pure CLPs are syntactically and semantically the same as CHC. The main difference is that sets of constrained Horn clauses are not necessarily intended for execution,

∗ The research leading to these results has received funding from the European Union 7th Framework Programme under grant agreement no. 318337, ENTRA - Whole-Systems Energy Transparency and the Danish Natural Science Research Council grant NUSA: Numerical and Symbolic Abstractions for Software Model Checking.

but rather as specifications. From the point of view of verification, we do not distinguish between CHC and pure CLP. Much research has been carried out on the analysis and transformation of CLP programs, typically for synthesising efficient programs or for inferring run-time properties of programs for the purpose of debugging, compile-time optimisations or program understanding. In this paper we investigate the application of this research to the CHC verification problem.

In Section 2 we define the CHC verification problem. In Section 3 we define a number of basic transformation and analysis components drawn from or inspired by the CLP literature. Section 4 discusses the role of these components in verification, illustrating them on an example problem. In Section 5 we construct a tool-chain out of these components and test it on a range of CHC verification benchmark problems. The results reported in this section represent one of the main contributions of this work. In Section 6 we propose possible extensions of the basic tool-chain and compare them with related work on CHC verification tool architectures. Finally in Section 7 we summarise the conclusions from this work.

2 Background: The CHC Verification Problem

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k) \rightarrow H(X))$ $(k \geq 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $B_1, \ldots, B_k, H$ are predicate symbols, $H(X)$ is the head of the clause and $\phi \wedge B_1(X_1) \wedge \ldots \wedge B_k(X_k)$ is the body. Sometimes the clause is written $H(X) \leftarrow \phi \wedge B_1(X_1), \ldots, B_k(X_k)$ and in concrete examples it is written in the form `H :-` ϕ, $\mathtt{B}_1\mathtt{(X}_1\mathtt{)}, \ldots, \mathtt{B}_k\mathtt{(X}_k\mathtt{)}$. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

We assume here that the constraint theory is closed with respect to negation and that there is a distinguished predicate symbol false which is interpreted as false. In practice the predicate false only occurs in the head of clauses; we call clauses whose head is false *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$ is equivalent to the formula $\mathsf{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge B_1(X_1), \ldots, B_k(X_k)$. The latter might not be a CHC but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set of CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be viewed as safety properties. If a set of CHCs encodes the behaviour of some system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHC clauses.

The CHC verification problem. To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. In every interpretation of P, the predicate false is interpreted as false; hence $P \models \mathsf{false}$ if and only if P has no model. Equivalently P has a model if and only if $P \not\models \mathsf{false}$. It is clear that any model of P assigns false to the bodies of integrity constraints.

The verification problem can be formulated deductively rather than model-theoretically. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure.

If the proof procedure is sound and complete then $P \not\models A$ if and only if $P \not\vdash A$. So the verification problem is to show (using CLP terminology) that the computation of the goal $\leftarrow \mathsf{false}$ in program P does not succeed using a complete proof procedure. Although in this work we follow the model-based formulation of the problem, we exploit the equivalence with the deductive formulation, which underlies, for example, the query-answer transformation and specialisation techniques to be presented.

2.1 Representation of Interpretations

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathcal{C}$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and $\mathcal{C}$ is a constraint over $Z_1, \ldots, Z_n$. If $\mathcal{C}$ is true we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathcal{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathcal{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

Minimal models. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator, $T_P^{\mathcal{C}}$, which is an extension of the standard T_P operator from logic programming, extended to handle constraints. Furthermore $\mathsf{lfp}(T_P^{\mathcal{C}})$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$. For more details, see (Jaffar and Maher 1994). This sequence provides a basis for abstract interpretation of CHC clauses.

Proof by over-approximation of the minimal model. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(v_1, \ldots, v_n)$ if and only if $p(v_1, \ldots, v_n) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathsf{false} \not\in M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathsf{false} \not\in M'$. This technique is called proof by over-approximation of the minimal model.

3 Relevant tools for CHC Verification

In this section, we give a brief description of some relevant tools borrowed from the literature in analysis and transformation of CLP.

Unfolding. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathcal{B}_1, p(Y), \mathcal{B}_2$ where $\mathcal{B}_1, \mathcal{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathcal{D}_i$, where the variables of these clauses are standardised apart from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \ldots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathcal{B}_1, Y = Z_i, \mathcal{D}_i, \mathcal{B}_2$. The equality $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i. It is a standard result that unfolding a clause in P preserves P's minimal model (Pettorossi and Proietti 1999). In particular, $P \models \mathsf{false} \equiv P' \models \mathsf{false}$.

Specialisation. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P, for instance, parts of P that are irrelevant to A could be omitted in P_A. In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A. Partial evaluation is a well-developed method (Gallagher 1993; Leuschel 1999).

We make use a form of specialisation know as forward slicing, more specifically redundant argument filtering (Leuschel and Sørensen 1996), in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A, denote by P_A^{raf} the program obtained by applying the RAF algorithm from (Leuschel and Sørensen 1996) with respect to the goal A. We have the property that $P \models A \equiv P_A^{\mathsf{raf}} \models A$ and in particular that $P \models \mathsf{false} \equiv P_{\mathsf{false}}^{\mathsf{raf}} \models \mathsf{false}$.

Query-answer transformation. Given a set of CHCs P and an atomic query A, the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P, using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p, two new predicates p_{ans} and p_{query} are defined. For an atomic formula A, A_{ans} and A_{query} denote the replacement of A's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and query A, the idea is to derive a program P_A^{qa} with the following property $P \models A$ iff $P_A^{\mathsf{qa}} \models A_{ans}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see (Debray and Ramakrishnan 1994; Gallagher and de Waal 1993; Codish and Demoen 1993). In particular, $P_{\mathsf{false}}^{\mathsf{qa}} \models \mathsf{false}_{ans} \equiv P \models \mathsf{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \mathsf{false}$.

Predicate splitting. Let P be a set of CHCs and let $\{c_1, \ldots, c_m\}$ be the set of clauses in P having some given predicate p in the head, where $c_i = p(X) \leftarrow \mathcal{D}_i$. Let $C_1, \ldots, C_k$ be some partition of $\{c_1, \ldots, c_m\}$, where $C_j = \{c_{j_1}, \ldots, c_{j_{n_j}}\}$. Define k new predicates $p_1 \ldots p_k$, where p_j is defined by the bodies of clauses in partition C_j, namely $D_j = \{p_j(X) \leftarrow \mathcal{D}_{j_1}, \ldots, p_j(X) \leftarrow \mathcal{D}_{j_{n_j}}\}$. Finally, define k clauses $C_p = \{p(X) \leftarrow p_1(X), \ldots, p(X) \leftarrow p_k(X)\}$. Then we define a splitting transformation as follows.

1. Let $P' = P \setminus \{c_1, \ldots, c_m\} \cup C_p \cup D^1 \cup \ldots \cup D^k$.
2. Let P^{split} be the result of unfolding every clause in P' whose body contains $p(Y)$ with the clauses C_p.

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses `new3(A,B) :- A=<99, new4(A,B)` and `new3(A,B) :- A>=100, new5(A,B)`, we produce two new predicates, since the constraints `A=<99` and `A>=100` are disjoint. The new predicates are defined by clauses `new3`$_1$`(A,B) :- A=<99, new4(A,B)` and `new3`$_2$`(A,B) :- A>=100, new5(A,B)`, and all calls to `new3` throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in (Serebrenik and De Schreye 2001). In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) (Pettorossi and Proietti 1999). Assuming that the predicate false is not split, we have that $P \models \mathsf{false} \equiv P^{\mathsf{split}} \models \mathsf{false}$.

Convex polyhedron approximation. Convex polyhedron analysis (Cousot and Halbwachs 1978) is a program analysis technique based on abstract interpretation (Cousot and Cousot 1977). When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathcal{C}$ for each predicate p. The constraint $\mathcal{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King (Benoy and King 1996). Since the domain of convex polyhedra contains infinite increasing chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One techniques is known as widening-upto, or widening with thresholds (Halbwachs et al. 1994). A threshold is an assertion that is combined with a widening operator to improve its precision. Recently, a technique for deriving more effective thresholds was developed (Lakhdar-Chaouch et al. 2011), which we have found to be effective in experimental studies.

4 The role of CLP tools in verification

The techniques discussed in the previous section play various roles. The convex polyhedron analysis, together with the helper tool to derive threshold constraints, constructs an approximation of the minimal model of a set of CHCs. If false (or false_{ans}) is not in the approximate model, then the verification problem is solved. Otherwise the problem is not solved; in effect a "don't know" answer is returned. We have found that polyhedron analysis alone is seldom precise enough to solve non-trivial CHC verification problems; in combination with the other tools, it is very effective.

Unfolding can improve the structure of a program, removing some case of mutual recursion, or propagating constraints upwards towards the integrity constraints, and can improve the precision and performance of convex polyhedron analysis.

Specialisation can remove parts of theories not relevant to the verification problem, and can also propagate constraint downwards from the integrity constraints. Both of these have a beneficial effect on performance and precision of polyhedron analysis.

Analysis of a query-answer program (with respect to false) is in effect the search for a derivation tree for false. Its effectiveness in CHC verification problems is variable. It can sometimes worsen performance since the query-answer transformed program is larger and contains more recursive dependencies than the original. On the other hand, one seldom loses precision and it is often more effective in allowing constraints to be propagated upwards (through the *ans* predicates) and downwards (through the *query* predicates).

4.1 Application of the tools

We illustrate the tools on a running example (Figure 1), one of the benchmark suite of the VeriMAP system (De Angelis et al. 2013). The result of applying unfolding is shown in

```
new6(A,B) :- B=<99.
new5(A,B) :- B>=101.
new5(A,B) :- B=<100, new6(A,B).
new4(A,B) :- C=1+A, A=<49, new3(C,B).
new4(A,B) :- C=1+A, D=1+B, A>=50, new3(C,D).
new3(A,B) :- A=<99, new4(A,B).
new3(A,B) :- A>=100, new5(A,B).
false :- A=0, B=50, new3(A,B).
```

Fig. 1. The example program `MAP-disj.c.map.pl`

```
false :- A=0, B=50, new3(A,B).
new3(A,B) :- A=<99, C = 1+A, A=<49, new3(C,B).
new3(A,B) :- A=<99, C = 1+A, D = 1+B, A>=50, new3(C,D).
new3(A,B) :- A>=100, B>=101.
new3(A,B) :- A>=100, B=<100, B=<99.
```

Fig. 2. Result of unfolding `MAP-disj.c.map.pl`

Figure 2 (omitting the definitions of the unfolded predicates `new4, new5` and `new6`, which are no longer reachable from `false`. The unfolding strategy we adopt is the following: the predicate dependency graph of a program consists of the set of edges (p, q) such that there is clause where p is the predicate of the head and q is a predicate occurring in the body. We perform a depth-first search of the predicate dependency graph, starting from `false`, and identify the backward edges, namely those edges (p, q) where q is an ancestor of p in the depth-first search. We then unfold every body call whose predicate is not at the end of a backward edge. In Figure 1, we thus unfold calls to `new4, new5` and `new6`.

The query-answer transformation is applied to the program in Figure 2, with respect to the goal `false` resulting in the program shown in Figure 3. The model of the predicate `new3_query` corresponds to those calls to `new3` that are reachable from the call in the integrity constraint. Explicit representation of the query predicates permits more effective propagation of constraints from the integrity clauses during model approximation.

The splitting transformation is now applied to the program in Figure 3. We do not show the complete program, which contains 30 clauses. Figure 4 shows the split definition of `new3_query`, which is split since the last two clauses for `new3_query` in Figure 3 have mutually disjoint constraints, when projected onto the head variables.

A convex polyhedron approximation is then computed for the split program, after computing threshold constraints for the predicates. The resulting approximate model is shown in Figure 5 as a set of constrained facts. Since the model does not contain any

```
false_ans :- false_query, A=0, B=50, new3_ans(A,B).
new3_ans(A,B) :- new3_query(A,B), A=<99, C = 1+A, A=<49, new3_ans(C,B).
new3_ans(A,B) :- new3_query(A,B),A=<99,C is 1+A,D is 1+B, A>=50, new3_ans(C,D).
new3_ans(A,B) :- new3_query(A,B), A>=100, B>=101.
new3_ans(A,B) :- new3_query(A,B), A>=100, B=<100, B=<99.
new3_query(A,B) :- false_query, A=0, B=50.
new3_query(A,B) :- new3_query(C,B), C=<99, A = 1+C, C=<49.
new3_query(A,B) :- new3_query(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
false_query.
```

Fig. 3. The query-answer transformed program for program of Fig. 2

```
new3_query___1(A,B) :- false_query___1, A=0, B=50.
new3_query___1(A,B) :- new3_query___1(C,B), C=<99, A = 1+C, C=<49.
new3_query___1(A,B) :- new3_query___2(C,B), C=<99, A = 1+C, C=<49.
new3_query___2(A,B) :- new3_query___1(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
new3_query___2(A,B) :- new3_query___2(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
```

Fig. 4. Part of the split program for the program in Fig. 3

```
false_query___1 :- []
new3_query___1(A,B) :- [1*A>=0,-1*A>= -50,1*B=50]
new3_query___2(A,B) :- [1*A>=51,-1*A>= -100,1*A+ -1*B=0]
```

Fig. 5. The convex polyhedral approximate model for the split program

constrained fact for `false_ans` we conclude that `false_ans` is not a consequence of the split program. Hence, applying the various correctness results for the unfolding, query-answer and splitting transformations, `false` is not a consequence of the original program.

Discussion of the example. Application of the convex polyhedron tool to the original, or the intermediate programs, does not solve the problem; all the transformations are needed in this case, apart from redundant argument filtering, which only affects efficiency.

The model of the query-answer program is finite for this example. However, the problem is essentially the same if the constants are scaled; for instance we could replace 50 by 5000, 49 by 4999, 100 by 10000 and 101 by 10001, and the problem is essentially unchanged. We noted that some CHC verification tools applied to this example solve the problem, but essentially by enumeration of the finite set of values encountered in the search. Such a solution does not scale well. On the other hand the polyhedral abstraction shown above is not an enumeration; an essentially similar polyhedron abstraction is generated for the scaled version of the example, in the same time. The VeriMAP tool (De Angelis et al. 2013) also handles the original and scaled versions of the example in the same time.

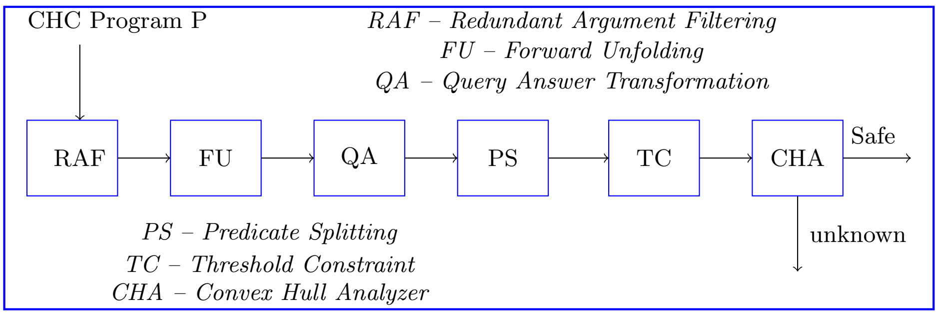

Fig. 6. *The basic tool chain for CHC verification.*

5 Combining off-the-shelf tools: Experiments

The motivation for our tool-chain, summarised in Fig. 6, comes from our example program, which is a simple yet challenging program. We applied the same tool-chain to a number of benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2[1] and other sources including (Gange et al. 2013) and some of the VeriMap benchmarks (De Angelis et al. 2013). Many of these problems are considered challenging because they cannot be solved by one or more of the state-of-the-art-verification tools. Programs taken from the SMT-LIB2 repository are first translated to CHC form. The results are summarised in Table 1.

In Table 1, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool-chain since they contain numbers which does not fit in 32 bits, the limit of our Ciao Prolog implementation. Whereas problems marked with (**) are solvable by our modified tool-chain. Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive required invariant. However overall result shows that our simple tool-chain begins to compete with advanced tools like HSF (Grebenshchikov et al. 2012), VeriMAP (De Angelis et al. 2013), TRACER (Gange et al. 2013), *etc.* We do not report timings, though all these results are obtained in a matter of seconds, since our tool-chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

Table 1. *Experiments results on CHC benchmark program*

SN	Program	Result	SN	Program	Result
1	MAP-disj.c.map.pl	verified	17	MAP-forward.c.map.pl	verified
2	pldi12.pl	verified	18	tridag.smt2	verified
3	t1.pl	verified	19	qrdcmp.smt2	verified
4	t1-a.pl	verified	20	choldc.smt2	verified
5	t2.pl	verified	21	lop.smt2	verified
6	t3.pl	verified	22	pzextr.smt2	verified
7	t4.pl	verified	23	qrsolv.smt2	verified
8	t5.pl	verified	24*	crank.smt2	NOT
9	MAP-disj.c.map-scaled.pl	verified	25*	bandec.smt2	NOT
10	INVGEN-id-build	verified	26**	amebsa.smt2	verified
11	INVGEN-nested5	verified	27	sshsimpl-s3-srvr-1b-safeil.c	NOT
12	INVGEN-nested6	verified	28*	sshsimpl-s3-srvr-1a-safeil.c	NOT
13	INVGEN-nested8	verified	29**	DAGGER-barbr.map.c	verified
14	INVGEN-svd-some-loop	verified	30	INVGEN-apache-escape-absolute	verified
15	INVGEN-svd1	verified	31	systemc-token-ring.01-safeil.c	NOT
16	INVGEN-svd4	verified	32	TRACER-testabs15	verified

[1] https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/

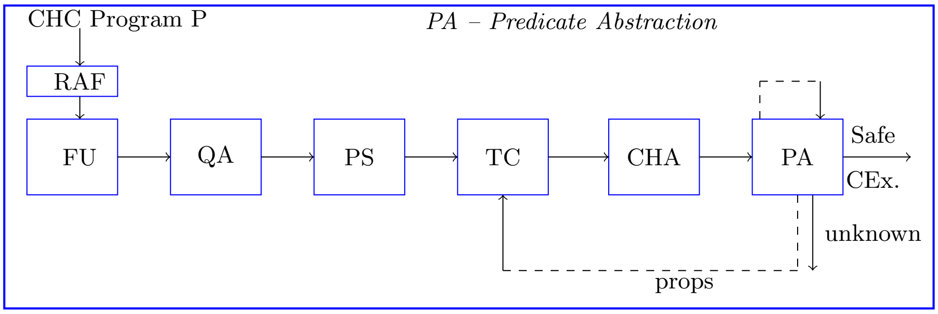

Fig. 7. *Future extension of our toolchain.*

6 Discussion and Related Work

The most similar work to ours is by De Angelis *et al.* (De Angelis et al. 2013) which is also based originally on CLP program transformation and specialisation. They construct a sequence of transformations of P, say, $P, P_1, P_2, \ldots, P_k$; essentially if P_k contains no clause with head false the problem is solved. A proof of unsafety is obtained if P_k contains a clause false $\leftarrow$. In contrast to their approach, we believe that our tool-chain-based approach gives more insight into the role of each transformation. Work by Gange et al. (Gange et al. 2013) is a top-town evaluation of CLP programs which records certain derivations and learns only from failed derivations. This helps to prune further derivations and helps to achieve termination in the presence of infinite executions. HSF(C) (Grebenshchikov et al. 2012) and Duality[2] are examples of the CEGAR approach (Counter-Example-Guided Abstraction Refinement). This approach can be viewed as property-based abstract interpretation based on a set of properties that is refined on each iteration. The refinement of the properties is the key problem in CEGAR; an abstract proof of unsafety is used to generate properties (often using interpolation) that prevents that proof from arising again. Thus more and more abstract counter-examples are successively eliminated. The relatively good performance of our tool-chain, without any refinement step at all, suggests that finding the right invariants is aided by a tool such as the convex polyhedron solver and the pre-processing steps we applied. In any case, property-based abstractions could easily be added to the tool-chain, perhaps using properties generated by the convex polyhedron solver. In Figure 7 we sketch possible extensions of our basic tool-chain, incorporating a refinement loop and property-based abstraction.

It should be noted that the query-answer transformation, predicate splitting and unfolding may all cause an increase in the program size. The convex polyhedron analysis becomes more effective as a result, but in the worst case the program size might restrict the applicability of the analysis. In future work, more sophisticated heuristics controlling these transformations, especially unfolding and splitting, are needed.

[2] http://research.microsoft.com/en-us/projects/duality/

7 Concluding remarks and future work

We have shown that a combination of off-the-shelf tools from CLP transformation and analysis, combined in a sensible way, is surprisingly effective in CHC verification. The component-based approach allowed us to experiment with the tool-chain until we found an effective combination. This experimentation is continuing and we are confident of making improvements by incorporating other standard techniques and by finding better heuristics for applying the tools. Further we would like to investigate the choice of chain suitable for each example since more complicated problems can be handled just by altering the chain. We also suspect from initial experiments that an advanced partial evaluator such as ECCE (Leuschel et al. 2006) will play a useful role. Our results give insights for further development of automatic CHC verification tools. We would like to combine our program transformation techniques with abstraction refinement techniques and experiment with the combination.

References

Benoy, F. and King, A. 1996. Inferring argument size relationships with CLP(R). In *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, J. P. Gallagher, Ed. Springer-Verlag Lecture Notes in Computer Science, vol. 1207. 204–223.

Codish, M. and Demoen, B. 1993. Analysing logic programs using "Prop"-ositional logic programs and a magic wand. In *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*, D. Miller, Ed. MIT Press.

Cousot, P. and Cousot, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds. ACM, 238–252.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*. 84–96.

De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. 2013. Verification of imperative programs by transforming constraint logic programs. In *CILC*, D. Cantone and M. N. Asmundo, Eds. CEUR Workshop Proceedings, vol. 1068. CEUR-WS.org, 83–98.

Debray, S. and Ramakrishnan, R. 1994. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming 18*, 149–176.

Gallagher, J. P. 1993. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, Copenhagen, 88–98.

Gallagher, J. P. and de Waal, D. 1993. Deletion of redundant unary type predicates from logic programs. In *Logic Program Synthesis and Transformation*, K. Lau and T. Clement, Eds. Workshops in Computing. Springer-Verlag, 151–167.

Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. 2013. Failure tabled constraint logic programming by interpolation. *TPLP 13,* 4-5, 593–607.

Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. 2012. Hsf(c): A software verifier based on Horn clauses - (competition contribution). In *TACAS*, C. Flanagan and B. König, Eds. LNCS, vol. 7214. Springer, 549–551.

Halbwachs, N., Proy, Y. E., and Raymound, P. 1994. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*. LNCS, vol. 864. 223–237.

Jaffar, J. and Maher, M. 1994. Constraint Logic Programming: A Survey. *Journal of Logic Programming 19/20*, 503–581.

Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. 2011. Widening with thresholds for programs with complex control graphs. In *ATVA 2011*, T. Bultan and P.-A. Hsiung, Eds. Lecture Notes in Computer Science, vol. 6996. Springer, 492–502.

Leuschel, M. 1999. Advanced logic program specialisation. In *Partial Evaluation - Practice and Theory*, J. Hatcliff, T. Æ. Mogensen, and P. Thiemann, Eds. Lecture Notes in Computer Science, vol. 1706. Springer, 271–292.

Leuschel, M., Elphick, D., Varea, M., Craig, S.-J., and Fontaine, M. 2006. The Ecce and Logen partial evaluators and their web interfaces. In *PEPM 2006*, J. Hatcliff and F. Tip, Eds. ACM, 88–94.

Leuschel, M. and Sørensen, M. H. 1996. Redundant argument filtering of logic programs. In *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, J. P. Gallagher, Ed. 83–103.

Pettorossi, A. and Proietti, M. 1999. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program. 41,* 2-3, 197–230.

Serebrenik, A. and De Schreye, D. 2001. Inference of termination conditions for numerical loops in Prolog. In *LPAR 2001*, R. Nieuwenhuis and A. Voronkov, Eds. Lecture Notes in Computer Science, vol. 2250. Springer, 654–668.