System Description: H-PILoT

Carsten Ihlemann and Viorica Sofronie-Stokkermans

Max-Planck-Institut für Informatik, Campus E 1.4, Saarbrücken, Germany

Abstract. This system description provides an overview of H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions), a program for hierarchical reasoning in extensions of logical theories with functions axiomatized by a set of clauses. H-PILoT reduces deduction problems in the theory extension to deduction problems in the base theory. Specialized provers and standard SMT solvers can be used for testing the satisfiability of the formulae obtained after the reduction. For local theory extensions this hierarchical reduction is sound and complete and – if the formulae obtained this way belong to a fragment decidable in the base theory – H-PILoT provides a decision procedure for testing satisfiability of ground formulae, and can also be used for model generation.

Key words: local theory extensions, hierarchical reasoning

1 Introduction

H-PILoT (Hierarchical Proving by Instantiation in Local Theory extensions) is an implementation of the method for hierarchical reasoning in local theory extensions presented in [6, 10, 12]: it reduces the task of checking the satisfiability of a (ground) formula over the extension of a theory with additional function symbols subject to certain axioms (a set of clauses) to the task of checking the satisfiability of a formula over the base theory. The idea is to replace the set of clauses which axiomatize the properties of the extension functions by a finite set of instances thereof. This reduction is polynomial in the size of the initial set of clauses and is always sound. It is complete in the case of so-called *local extensions* [10]; in this case, it provides a decision procedure for the universal theory of the theory extension if the clauses obtained by the hierarchical reduction belong to a fragment decidable in the base theory. The satisfiability of the reduced set of clauses is then checked with a specialized prover for the base theory.

State of the art SMT provers such as CVC3, Yices and Z3 [1, 5, 3] are very efficient for testing the satisfiability of *ground formulae* over standard theories, but use heuristics in the presence of *universally quantified* formulae, hence cannot detect *satisfiability* of such formulae. H-PILoT recognizes a class of local axiomatizations, performs the instantiation and hands in a ground problem to the SMT provers or other specialized provers, for which they are know to terminate with a yes/no answer, so it can be used as a tool for steering standard SMT provers, in order to provide decision procedures in the case of local theory extensions. H-PILoT can also be used for generating models of satisfiable

formulae; and even more, it can be coupled to programs with graphic facilities to provide graphical representations of these models. Being a decision procedure for many theories important in verification, H-PILoT is extremely helpful for deciding truth or satisfiability in a large variety of verification problems.

2 Description of the H-PILoT Implementation

H-PILoT is an implementation of the method for hierarchical reasoning in local theory extensions presented in [6, 10–12]. H-PILoT is implemented in Ocaml. The system (with manual and examples) can be downloaded from `www.mpi-inf.mpg.de/~ihlemann/software/`. Its general structure is presented in Figure 1.

2.1 Theoretical Background

Let $\mathcal{T}_0$ be a Σ_0-theory. We consider extensions $\mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ with function symbols in a set Σ_1 (extension functions) whose properties are axiomatized by a set $\mathcal{K}$ of $\Sigma_0 \cup \Sigma_1$-clauses. Let Σ_c be an additional set of constants.

Task. Let G be a set of ground $\Sigma_0 \cup \Sigma_1 \cup \Sigma_c$-clauses. We want to check whether or not G is satisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$.

Method. Let $\mathcal{K}[G]$ be the set of those instances of $\mathcal{K}$ in which every subterm starting with an extension function is a ground subterm already appearing in $\mathcal{K}$ or G. If G is unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}[G]$ then it is also unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$. The converse is not necessarily true. We say that the extension $\mathcal{T}_0 \cup \mathcal{K}$ of $\mathcal{T}_0$ is *local* if for each set G of ground clauses, G is unsatisfiable w.r.t. $\mathcal{T}_0 \cup \mathcal{K}$ if and only if $\mathcal{K}[G] \cup G$ has no partial $\Sigma_0 \cup \Sigma_1$-model whose Σ_0-reduct is a total model of $\mathcal{T}_0$ and in which all Σ_1-terms of $\mathcal{K}$ and G are defined.

Theorem 1 ([10]). *Assume that the extension $\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}$ is local and let G be a set of ground clauses. Let $\mathcal{K}^0 \cup G^0 \cup D$ be the purified form of $\mathcal{K} \cup G$ obtained by introducing fresh constants for the Σ_1-terms, adding their definitions $d \approx f(t)$ to D, and replacing $f(t)$ in G and $\mathcal{K}[G]$ by d. (Then Σ_1-functions occur only in D in unit clauses of the form $d \approx f(t)$.) The following are equivalent.*

1. *$\mathcal{T}_0 \cup \mathcal{K} \cup G$ has a (total) model.*
2. *$\mathcal{T}_0 \cup \mathcal{K}[G] \cup G$ has a partial model where all subterms of $\mathcal{K}$ and G and all Σ_0-functions are defined.*
3. *$\mathcal{T}_0 \cup \mathcal{K}^0 \cup G^0 \cup \mathsf{Con}^0$ has a total model, where $\mathsf{Con}^0 := \{ \bigwedge_{i=1}^n c_i \approx d_i \rightarrow c \approx d \mid f(c_1, ..., c_n) \approx c, f(d_1, ..., d_n) \approx d \in D \}$.*

A variant of this notion, namely Ψ-locality, was also studied, where the set of instances to be taken into account is $\mathcal{K}[\Psi(G)]$, where Ψ is a closure operator which may add a (finite) number of new terms to the subterms of G. We also analyzed a generalized version of locality, in which the clauses in $\mathcal{K}$ and the set G of ground clauses are allowed to contain first-order Σ_0-formulae.

Examples of Local Extensions. Among the theory extensions which we proved to be local or Ψ-local in previous work are:

- theories of pointers with stored scalar information in the nodes [8, 7];
- theories of arrays with integer indices, and elements in a given theory [2, 7];
- theories of functions (e.g. over an ordered domain, or over a numerical domain) satisfying e.g. monotonicity or boundedness conditions [10, 14];
- various combinations of such extensions [12, 7].

We can also consider successive extensions of theories: $\mathcal{T}_0 \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \subseteq \cdots \subseteq \mathcal{T}_0 \cup \mathcal{K}_1 \cup \cdots \cup \mathcal{K}_n$. If every variable in $\mathcal{K}_i$ occurs below a function symbol in Σ_i, this reduction process can be iterated [7].

2.2 Preprocessing

H-PILoT receives as input a many-sorted specification of the signature; a specification of the hierarchy of local extensions to be considered; an axiomatization $\mathcal{K}$ of the theory extension(s); a set G of ground clauses containing possibly additional constants. H-PILoT allows the following pre-processing functionality:

Translation to Clause Form. H-PILoT provides a translator to clause normal form (CNF) for ease of use. First-order formulas can be given as input; H-PILoT translates them into CNF.[1]

Flattening/Linearization. Methods for recognizing local theory extensions usually require that the clauses in the set $\mathcal{K}$ extending the base theory are *flat* and *linear*[2]. If the flags -linearize and/or -flatten are used then the input is flattened and/or linearized. H-PILoT allows the user to enter a more readable non-flattened version and will perform the flattening and linearization of $\mathcal{K}$.

Recognizing Syntactic Criteria which Imply Locality. Examples of local extensions include those mentioned in Section 2.1 (and also iterations and combinations thereof). In the pre-processing phase H-PILoT analyzes the input clauses to check whether they are in one of these fragments.

- If the flag - array is on: checks if the input is in the array property fragment[2];
- if the keyword "pointer" is detected: checks if the input is in the pointer fragment in [8] and possibly adds premises of the form "$t \neq \mathsf{null}$".

If the answer is "yes", we know that the extensions we consider are local, i.e. that H-PILoT can be used as a decision procedure. We are currently extending the procedure to recognize "free", "monotone" and "bounded" functions.

[1] In the present implementation, the CNF translator does not provide the full functionality of FLOTTER [9], but is powerful enough for most applications. (An optimized CNF translator is being implemented.)

[2] Flatness means that extension functions may not be nested; linearity means that variables may occur only in the same extension term (which may appear repeatedly).

2.3 Main Algorithm.

The main algorithm hierarchically reduces a decision problem in a theory extension to a decision problem in the base theory.

Given a set of clauses $\mathcal{K}$ and a ground formula G, the algorithm we use carries out a hierarchical reduction of G to a set of formulae in the base theory, cf. Theorem 1. It then hands over the new problem to a dedicated prover such as Yices, CVC3 or Z3. H-PILoT is also coupled with Redlog (for handling non-linear real arithmetic) and with SPASS[3].

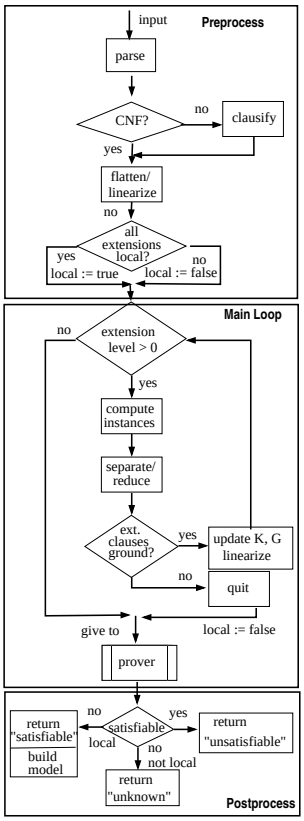

Fig. 1. H-PILoT Structure

Loop. For a chain of local extensions:

$$\mathcal{T}_0 \subseteq \mathcal{T}_1 = \mathcal{T}_0 \cup \mathcal{K}_1 \subseteq \mathcal{T}_2 = \mathcal{T}_0 \cup \mathcal{K}_1 \cup \mathcal{K}_2 \subseteq ... \subseteq \mathcal{T}_n = \mathcal{T}_0 \cup \mathcal{K}_1 \cup ... \cup \mathcal{K}_n.$$

a satisfiability check w.r.t. the last extension can be reduced (in n steps) to a satisfiability check w.r.t. $\mathcal{T}_0$. The only caveat is that at each step the reduced clauses $\mathcal{K}_i^0 \cup G^0 \cup \mathsf{Con}^0$ need to be ground. Groundness is assured if each variable in a clause appears at least once under an extension function. In that case, we know that at each reduction step the total clause size only grows polynomially in the size of $\Psi(G)$ [10]. H-PILoT allows the user to specify a chain of extensions by tagging the extension functions with their place in the chain (e.g., if f belongs to $\mathcal{K}_3$ but not to $\mathcal{K}_1 \cup \mathcal{K}_2$ it is declared as level 3). Let $i = n$. As long as the extension level $i > 0$, we compute $\mathcal{K}_i[G]$ ($\mathcal{K}_i[\Psi(G)]$ in case of arrays). If no separation of the extension symbols is required, we stop here (the result will be passed to an external prover). Otherwise, we perform the hierarchical reduction in Theorem 1 by purifying $\mathcal{K}_i$ and G (to $\mathcal{K}_i^0, G^0$ resp.) and by adding corresponding instances of the congruence axioms Con_i. To prepare for the next iteration, we transform the clauses into the form $\forall x.\Phi \vee \mathcal{K}_i$ (compute prenex form, skolemize). If $\mathcal{K}_i[G]/\mathcal{K}_i^0$ is not ground, we quit with a corresponding message. Otherwise we set: $G' := \mathcal{K}_i^0 \wedge G^0 \wedge \mathsf{Con}_i, \mathcal{T}_0' := \mathcal{T}_0 \setminus \{\mathcal{K}_{i-1}\}, \mathcal{K}' := \mathcal{K}_{i-1}$. We flatten and linearize $\mathcal{K}'$ and decrease i. If level $i = 0$ is reached, we exit the main loop and G' is handed to an external prover[4].

[3] H-PILoT only calls one of these solvers once.

[4] Completeness is guaranteed if all extensions are known to be local and if each reduction step produces a set of ground clauses for the next step.

2.4 Post-processing

Depending on the answer of the external provers to the satisfiability problem G_n, we can infer whether the initial set G of clauses was satisfiable or not.

- If G_n is unsatisfiable w.r.t. $\mathcal{T}_0$ then we know that G is unsatisfiable.
- If G_n is satisfiable, but H-PILoT failed to detect, and the user did not assert the locality of the sets of clauses used in the axiomatization, its answer is "unknown".
- If G_n is satisfiable and H-PILoT detected the locality of the axiomatization, then the answer is "satisfiable". In this case, H-PILoT takes advantage of the ability of SMT-solvers to provide counter-examples for the satisfiable set G_n of ground clauses and specifies a counter-example of G by translating the basic SMT-model of the reduced query to a model of the original query[5]. The counter-examples can be graphically displayed using Mathematica. This is currently done separately; an integration with Mathematica is planned for the future.

3 System Evaluation

We have used H-PILoT on a variety of local extensions and on chains of local extensions. The flags that we used are described in the manual (see `www.mpi-inf.mpg.de/~ihlemann/software/`). An overview of the tests we made is given in sections 3.1 and 3.2. In analyzing them, we distinguish between satisfiable and unsatisfiable problems.

Unsatisfiable Problems. For simple unsatisfiable problems, there hardly is any difference in run-time whether one uses H-PILoT or an SMT-solver directly.[6] When we consider chains of extensions the picture changes dramatically. On one test example (array insert), checking an invariant of an array insertion algorithm, which used a chain of two local extensions, Yices performed considerably slower than H-PILoT: The original problem took Yices 318.22s to solve. The hierarchical reduction yielded 113 clauses of the background theory (integers), proved unsatisfiable by Yices in 0.07s.

Satisfiable Problems. For satisfiable problems over local theory extensions, H-PILoT always provides the right answer. In local extensions, H-PILoT is a decision procedure whereas completeness of other SMT-solvers is not guaranteed. In the test runs, Yices either ran out of memory or took more than 6 hours when given any of the unreduced problems. This even was the case for small problems, e.g. problems over the reals with less than ten clauses. With H-PILoT as a front end, Yices solved all the satisfiable problems in less than a second.

[5] This improves readability greatly, especially when we have a chain of extensions.

[6] This is due to the fact that a good SMT-solver uses the heuristic of trying out all the occurring ground terms as instantiations of universal quantifiers. For local extensions this is always sufficient to derive a contradiction.

3.1 Test runs for H-PILoT

We analyzed the following examples[7]:

array insert. Insertion of an element into a sorted integer array. Arrays are definitional extensions here.

array insert (∃). Insertion of an element into a sorted integer array. Arrays are definitional extensions here. Alternate version with (implicit) existential quantifier.

array insert (linear). Linear version of **array insert**.

array insert real. Like **array insert** but with an array of reals.

array insert real (linear). Linear version of **array insert real**.

update process priorities (∀∃). Updating of priorities of processes. This is an example of extended locality: We have a ∀∃-clause.

list1. Made up example of integer lists. Some arithmetic is required

chain1. Simple test for chains of extensions (plus transitivity).

chain2. Simple test for chains of extensions (plus transitivity and arithmetic).

double array insert. Problem of the array property fragment [2]. (A sorted array is updated twice.)

mono. Two monotone functions over integers/reals for SMT solver.

mono for distributive lattices.R. Two monotone functions over a distributive lattice. The axioms for a distributive lattice are stated together with the definition of a relation R: $R(x, y) :\Leftrightarrow x \wedge y = x$. Monotonicity of f (resp. of g) is given in terms of R: $R(x, y) \rightarrow R(f(x), f(y))$. Flag `-freeType` must be used.

mono for distributive lattices. Same as **mono for distributive lattices.R** except that no relation R is defined. Monotonicity of the two functions f, g is directly given: $x \wedge y = x \rightarrow f(x) \wedge f(y) = f(x)$. (Much harder than defining R.) Flag `-freeType` must be used.

mono for poset. Two monotone functions over a poset with poset axioms. Same as **mono**, except the order modeled by a relation R. Flag `-freeType` should be used.

mono for total order. Same as **mono** except linearity is an axiom. This makes no difference unless SPASS is used.

own. Simple test for monotone function.

mvLogic/mv1.sat. Example for MV-algebras. The Łukasiewicz connectives can be defined in terms of the (real) operations $+, -, \leq$. Linearity is deducible from axioms.

mvLogic/mv2. Example for MV-algebras. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$. The BL axiom is deducible.

mvLogic/bl1. Example for MV-algebras with BL axiom (redundantly) included. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$.

mvLogic/example_6.1. Example for MV-algebras with monotone and bounded function. The Łukasiewicz connectives can be defined in terms of $+, -, \leq$.

RBC_simple. Example with train controller.

RBC_variable2. Example with train controller.

[7] The satisfiable variant of a problem carries the suffix ".sat".

3.2 Test results

Name	**status**	**#cl.**	**H-PILoT + yices**	**H-PILoT + yices** stop at $\mathcal{K}[G]$	**yices**
array insert (implicit ∃)	Unsat	310	0.29	0.06	0.36
array insert (implicit ∃).sat	Sat	196	0.13	0.04	time out
array insert	Unsat	113	0.07	0.03	318.22[1]
array insert (linear version)	Unsat	113	0.07	0.03	7970.53[2]
array insert.sat	Sat	111	0.07	0.03	time out
array insert real	Unsat	113	0.07	0.03	360.00[1]
array insert real (linear)	Unsat	113	0.07	0.03	7930.00[2]
array insert real.sat	Sat	111	0.07	0.03	time out
update process priorities	Unsat	45	0.02	0.02	0.03
update process priorities.sat	Sat	37	0.02	0.02	unkown
list1	Unsat	18	0.02	0.01	0.02
list1.sat	Sat	18	0.02	0.01	unknown
chain1	Unsat	22	0.01	0.01	0.02
chain2	Unsat	46	0.02	0.02	0.02
mono	Unsat	20	0.01	0.01	0.01
mono.sat	Sat	20	0.01	0.01	unknown
mono for distributive lattices.R	Unsat	27	0.22	0.06	0.03
mono for distributive lattices.R.sat	Sat	27	unknown*	unknown*	unknown
mono for distributive lattices	Unsat	17	0.01	0.01	0.02
mono for distributive lattices.sat	Sat	17	0.01	0.01	unknown
mono for poset	Unsat	20	0.02	0.02	0.02
mono for poset.sat	Sat	20	unknown*	unknown*	unknown
mono for total order	Unsat	20	0.02	0.02	0.02
own	Unsat	16	0.01	0.01	0.01
mvLogic/mv1	Unsat	10	0.01	0.01	0.02
mvLogic/mv1.sat	Sat	8	0.01	0.01	unknown
mvLogic/mv2	Unsat	8	0.01	0.01	0.06
mvLogic/bl1	Unsat	22	0.02	0.01	0.03
mvLogic/example_6.1	Unsat	10	0.01	0.01	0.03
mvLogic/example_6.1.sat	Sat	10	0.01	0.01	unknown
RBC_simple	Unsat	42	0.03	0.02	0.03
double array insert	Unsat	606	1.16	0.20	0.07
double array insert	Sat	605	1.10	0.20	unknown
RBC_simple.sat	Sat	40	0.03	0.02	out. mem.
RBC_variable2	Unsat	137	0.08	0.04	0.04
RBC_variable2.sat	Sat	136	0.08	0.04	out. mem.

User + sys times (in s). Run on an Intel Xeon 3 GHz, 512 K-byte cache. Median of 100 runs (entries marked with [1]: 10 runs; marked with [2]: 3 runs). The third column lists the number of clauses produced; "unknown" means Yices answer was "unknown", "out. mem." means out of memory and time out was set at 6h. For an explanation of "unknown*" see below.

(*) The answer "unknown" for the satisfiable example with monotone functions over distributive lattices/posets (H-Pilot followed by Yices) is due to the fact that Yices

cannot handle the universal axioms of distributive lattices/posets. A translation of such problems to SAT provides a decision procedure (cf. [10] and also [15]).

Acknowledgments. This work was partly supported by the German Research Council (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS, `www.avacs.org`).

References

1. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 298–302. Springer, Heidelberg (2007)
2. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
3. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: TACAS 2008. LNCS, vol. 4963, pp. 337–340, Springer, Heidelberg (2008)
4. Dolzmann, A., Sturm, T.: Redlog: Computer algebra meets computer logic. In: ACM SIGSAM Bulletin, vol. 31(2), pp. 2–9 (1997)
5. Dutertre, B., de Moura, L.: Integrating Simplex with DPLL(T). CSL Technical Report, SRI-CSL-06-01 (2006)
6. Ganzinger, H.: Relating semantic and proof-theoretic concepts for polynomial time decidability of uniform word problems. In: 16th IEEE Symposion on Logic in Computer Science, pp. 81–92. IEEE press, New York (2001)
7. Ihlemann, C., Jacobs, S., Sofronie-Stokkermans, V.: On local reasoning in verification. In: TACAS 2008, LNCS, vol. 4963, pp. 265–281. Springer, Heidelberg (2008)
8. McPeak, S., Necula, G.C.: Data structure specifications via local equality axioms. In: CAV 2005, LNCS, vol. 3576, pp. 476–490. Springer, Heidelberg (2005)
9. Nonnengart, A., Weidenbach, C.: Computing Small Clause Normal Forms. In: Robinson, A. Voronkov, A. (eds.) Handbook of Automated Reasoning. pp. 335 – 367, Elsevier, Amsterdam (2001)
10. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE-20. LNAI, vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
11. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: IJCAR 2006. LNAI, vol. 4130, pp. 235–250. Springer, Heidelberg (2006)
12. Sofronie-Stokkermans, V.: Hierarchical and modular reasoning in complex theories: The case of local theory extensions. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS, vol. 4720, pp. 47–71. Springer, Heidelberg (2007)
13. Sofronie-Stokkermans, V.: Efficient Hierarchical Reasoning about Functions over Numerical Domains. In: KI 2008. LNAI, vol. 5243, pp. 135–143. Springer, Heidelberg (2008)
14. Sofronie-Stokkermans, V., Ihlemann, C.: Automated reasoning in some local extensions of ordered structures. J. Multiple-Valued Logics and Soft Computing 13(4–6), 397–414 (2007)
15. Sofronie-Stokkermans, V.: Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators. J. Symb. Comp. 36(6), 891–924 (2003)