Abstraction-Based Partial Deduction for Solving Inverse Problems – A Transformational Approach to Software Verification
(Extended Abstract)

Robert Glück[1] and Michael Leuschel[2]

[1] DIKU, Department of Computer Science,
University of Copenhagen, DK-2100 Copenhagen, Denmark
Email: glueck@diku.dk
[2] Department of Electronics and Computer Science
University of Southampton, Southampton SO17 1BJ, UK
Email: mal@ecs.soton.ac.uk

Abstract. We present an approach to software verification by program inversion, exploiting recent progress in the field of automatic program transformation, partial deduction and abstract interpretation. Abstraction-based partial deduction can work on infinite state spaces and produce finite representations of infinite solution sets. We illustrate the potential of this approach for infinite model checking of safety properties.

1 Introduction

Modern computing applications increasingly require software and hardware systems that are extremely reliable. Unfortunately, current validation techniques are often unable to provide high levels of assurance of correctness either due to the size and complexity of these systems, or because of fundamental limitations in reasoning about a given system. This paper examines the latter point showing that *abstraction-based partial deduction* can serve as a powerful analytical tool. This has several advantages in comparison with, e.g., standard logic programming. Among others, abstraction-based partial deduction has the ability to form recursively defined answers and can be used for program verification.

We apply the inversion capabilities of abstraction-based partial deduction to other languages using interpretive definitions. This means that a wide class of different verification tasks can be analyzed in a common framework using a set of uniform transformation techniques. We examine the potential for infinite model checking, and support our claims by several computer experiments.

2 Inversion, Partial Deduction, and Interpreters

Inversion While direct computation is the calculation of the output of a program for a given input, *inverse computation* is the calculation of the possible

D. Bjørner, M. Broy, A.V. Zamulin (Eds.): Perspectives of System Informatics. Proceedings.
LNCS 1755, pp. 93–100, 2000.

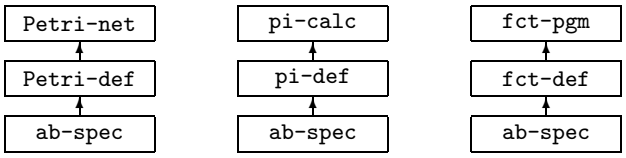


Fig. 1. Abstraction-based partial deduction `ab-spec` applied to Petri-nets, π-calculus, and functional programs via interpretive language definitions

input of a program for a given output. Consider the familiar *append* program, it can be run forwards (to concatenate two lists) and backwards (to split a list into sublists). Advances in this direction have been made in logic programming, based on solutions emerging from logic and proof theory.

However, inversion problems are not restricted to logic languages. Reasoning about the correctness of, say, a software specification, one may need to verify whether and how a critical state can be reached from any *earlier* state. This analysis requires inverse computation. The key idea is this: to show that a given system satisfies a given specification—representing a safety property—start with the bad states violating the specification, work *backwards* and show that no initial state leads to such a bad state.

Abstraction-Based Partial Deduction The relationship between *abstract interpretation* and *program specialisation* has been observed and several formal frameworks have been developed [6, 10, 9]. *Abstraction-based partial deduction* (APD) combines these two approaches and can thereby solve specialisation and analysis tasks which are outside the reach of either method alone [12, 11]. It was shown that program specialisation combined with abstract interpretation can vastly improve the power of both techniques (e.g., going beyond regular approximations or set-based analysis) [12].

Interpreters Language-independence can be achieved through the *interpretive approach* [18, 7, 1]: an interpreter serves as mediator between a (domain-specific) language and the language for which the program transformer is defined. Efficient implementations can be obtained by removing the interpretive overhead using program specialisation (a notable example are the Futamura projections). Work on porting inverse computation to new languages includes the inversion of imperative programs by treating their relational semantics as logic programs [15] and applying the Universal Resolving Algorithm to interpreters written in a functional language [1].

Our Approach The approach we will pursue in this paper is twofold. First, we apply the power of APD to inverse computation tasks. Instead of enumerating a list of substitutions, as in logic programming, we produce a *new logic program* which can be viewed as model of the original program instantiated to the given query. The transformation will (hopefully) derive a much simpler program (such as `p :- fail`), but APD has also the ability to form recursively defined programs.

Second, we use the interpretive approach to achieve language-independence. APD is implemented for a logic language, but we can apply its inversion capabilities to different language paradigms, such as Petri-nets and the π-calculus, via interpreters without having to write tools for each language (see Fig. 1).

To put these ideas to a trial, we use the ECCE logic program specialiser [12, 13]—employing *advanced control techniques* such as characteristic trees to guide the specialisation process—coupled with an *abstract interpretation technique*. (A more detailed technical account is beyond the scope of this extended abstract; the interested reader will find a complete description in [12, 13].) This APD-system does not yet implement the full power of [12, 11], but it will turn out to be sufficiently powerful for our purposes.

3 Advanced Inversion Tasks for Logic Programs

To illustrate three questions about a software requirement specification relying on solving inversion problems, let us consider a familiar example: exponentiation of natural numbers ($z = x^y$).

1. *Existence of solution?* Given output state z (e.g. $z = 3$), does there exist an input state x, y with $y > 1$ that gives raise to z? *Answer:* state $z = 3$ can never be reached. Observe that here we are not interested in the values of x, y, we just want to know whether such values exist. We will call such a setting *inversion checking*.
2. *Finiteness of solution?* Given output state z (e.g. $z = 4$), is there a finite number of input states x, y that can give raise to z? *Answer:* only two states ($x = 4, y = 1$ and $x = 2, y = 2$) lead to $z = 4$.
3. *Finite description of infinite solution?* Given output state z (e.g. $z = 1$), can an infinite set of input states be described in a finite form? *Answer:* any input state with $y = 0$ leads to $z = 1$, regardless of x.

Example 1. All three questions from above can be answered with APD. Consider a logic program encoding exponentiation of natural numbers where numbers are represented by terms of type $\tau = \texttt{0} \mid \texttt{s(}\tau\texttt{)}$.

```
exp(Base,0,s(0)).
exp(Base,s(Exp),Res) :- exp(Base,Exp,BE),mul(BE,Base,Res).
mul(0,X,0).
mul(s(X),Y,Z) :- mul(X,Y,XY),plus(XY,Y,Z).
plus(0,X,X).
plus(s(X),Y,s(Z)) :- plus(X,Y,Z).
```

1. *Existence of solution.* Inverting the program for $x^y = 3, y > 1$, that is by specialising `exp/2` wrt. goal `exp(X,s(s(Y)),s(s(s(0))))`, produces an empty program: no solution exists.
2. *Finiteness of solution.* Inverting $x^y = 4$, that is by specialising `exp/2` wrt. goal `exp(X,Y,s(s(s(s(0)))))`, produces a program in which the two solutions $x = 4, y = 1$ and $x = 2, y = 2$ are explicit:

```
exp__1(s(s(s(s(0)))),s(0)).
exp__1(s(s(0)),s(s(0))).
```

3. *Finite representation of infinite solution.* Finally, inverting $x^y = 1$ can be solved by specialising `exp/2` wrt. goal `exp(X,Y,s(0))`. The result is a recursive program: infinitely many solutions were found (x^0 and 1^y for any x, y) and described in a *finite* way.[1] This finite description is possible in our approach, but not in conventional logic programming, because APD generates (recursive) programs instead of enumerating an (infinite) list of answers.

```
exp__1(X1,0).
exp__1(s(0),s(X1)) :- exp_conj__2(X1).
exp_conj__2(0).
exp_conj__2(s(X1)) :- exp_conj__3(X1).
exp_conj__3(0).
exp_conj__3(s(X1)) :- exp_conj__3(X1).
```

Example 2. As a more practical application, take the following program which allows to determine whether a list has an even number of elements (`pairl/1`) and to delete from a list an element contained in the list (`del/3`).

```
pairl([]).
pairl([A|X]):- oddl(X).
oddl([A|X]):- pairl(X).

del(X,[X|T],T).
del(X,[Y|T],[Y|DT]):- X\=Y,del(X,T,DT).
```

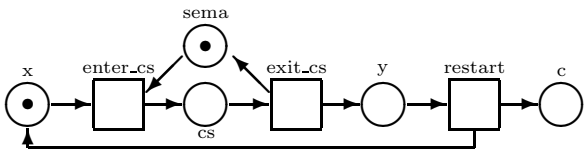
One might want to verify the property that deleting an element from a pair list will not result in a pair list. This can be translated into requiring that the following predicate always fails: `error(X,L) :- pairl(L),del(X,L,DL),pairl(DL).` which *can* be deduced by our APD-system: `error__1(X,L) :- fail.` To conclude, APD can invert programs in ways not possible with other approaches.

4 Case Study: Inversion and Infinite Model Checking

Recent years have seen considerable growth [5] in the application of model checking techniques [4, 3] to the validation and verification of correctness properties of hardware, and more recently software systems. The method is to model a hardware or software system as a finite, labelled transition system (LTS) which is then exhaustively explored to decide whether a given specification holds for all reachable states. One can even use tabling-based logic programming as an efficient means of performing explicit model checking [14].

However, many software systems cannot be modelled by a *finite* LTS (or similar system) and, as a consequence, there has been a lot of effort to enable *infinite model checking* (e.g., [17]). We argue that inverse computation in general, and our APD-technique in particular, has a lot to offer for this avenue of research:

[1] The residual program can be improved by better post-processing.




```
start(Trace,X,ReachableMarking) :-
       trace(Trace,[X,s(0),0,0,0],ReachableMarking).
trace([],Marking,Marking).
trace([Action|As],InMarking,OutMarking) :-
       trans(Action,InMarking,M1),trace(As,M1,OutMarking).
trans(enter_cs,[s(X),s(Sema),CritSec,Y,C],[X,Sema,s(CritSec),Y,C]).
trans(exit_cs,[X,Sema,s(CritSec),Y,C],[X,s(Sema),CritSec,s(Y),C]).
trans(restart,[X,Sema,CritSec,s(Y),C],[s(X),Sema,CritSec,Y,s(C)]).
```

Fig. 2. Petri net with a single semaphore and its encoding as logic program

- The system to be verified can be modelled as a program (possibly by means of an interpreter). This obviously includes finite LTS but also allows to express systems with an infinite number of states.
- Model checking of safety properties then amounts to *inversion checking*: we prove that a specification holds by showing that there exists no trace (the input argument) which leads to an invalid state.
- To be successful, infinite model checking requires refined abstractions (a key problem mentioned in [5]). The control of generalisation of APD provides just that (at least for the examples we treated so far). In essence, the *specialisation component of APD performs a symbolic traversal of the state space, thereby producing a finite representation of it, on which the abstract interpretation performs the verification of the specification.*

Consider the Petri net shown in Fig. 2. It models a single process which may enter a critical section (cs), the access to which is controlled by a semaphore (sema). The Petri net can be encoded directly as a logic program using an interpreter for Petri-nets `trace/3`, where the object-level Petri net is represented by `trans/3` facts. The `trace/3` predicate checks for enabled transitions and fires them.

The initial marking of `trace/3` can be seen in `start/3`: 1 token in the semaphore (sema), no tokens in the reset counter (c), no processes in the critical section (cs) and no processes in the final place (y). There may be `X` processes in the initial place (x). Again, numbers are represented by terms of type τ = `0` | `s(`τ`)`. More processes can be modelled if we increase the number of tokens in the initial place (x). Forward execution of the Petri net: given an initial value for `X` and a sequence of transitions `trace` determine the marking(s) that can be reached.

Let us now check a safety property of the given Petri net, namely that it is *impossible* to reach a marking where two processes are in their critical section at the same time. Clearly, this is an inversion task: given a marking where two

processes are in the critical section, try to find a trace that leads to this state. More precisely we want to do *inversion checking*, as the desired outcome is to prove that *no* inverse exists.

Example 3. Inverting the Petri net by specialising the interpreter in Fig. 2 wrt. the query `start(Tr,s(0),[X,S,s(s(CS)),Y,C])` we obtain the following program:

```
start(Tr,s(0),[X3,X4,s(s(X5)),X6,X7]) :- fail.
```

This inversion task cannot be solved by PROLOG (or XSB-PROLOG [16] with tabling), even when adding moding or delay declarations. Due to the counter (c) we have to perform infinite model checking which in turn requires *abstraction* and *symbolic execution*. Both of these are provided by our APD approach.

Example 4. Similarly, one can prove the safety property *regardless* of the number of processes, i.e., for *any* number of tokens in the initial place (x). When we specialise the interpreter of Fig. 2 for the query `unsafe(X,s(0),0,0,0)` we get (after 2 iterations each of the specialisation and abstract interpretation components of ECCE): `start(Tr,Processes,[X3,X4,s(s(X5)),X6,X7]) :- fail.`

5 Porting to other Languages and Paradigms

We can apply the power of our APD-approach, along with its capabilities for inversion and verification [8], to the π-calculus by writing an interpreter for it. We have also successfully ported inverse computation to a functional language via an interpreter (omitted from extended abstract). Apart from highlighting the power of our approach, these examples provide further computational evidence for the theoretical result [1] that inverse computation can applied to arbitrary languages via interpreters.

6 Conclusion and Assessment

We presented an approach to program inversion, exploiting progress in the field of automatic program transformation, partial deduction and abstract interpretation. We were able to port these inversion capabilities to other languages via interpretive definitions. We examined the potential for infinite model checking of safety properties, and supported our claims by computer experiments. We believe, by exploiting the connections between software verification and automatic program specialisation, one may be able to significantly extend the capabilities of analytical tools that inspect the input/output behaviour.

The emphasis was on novel ways of reasoning rather than efficiency and large scale applications. In principle, it is possible to extend our approach to verify larger, more complicated infinite systems.[2] As with all automatic specialisation

[2] Larger systems have been approached with related techniques as a processing phase [9]. However, their purpose is to reduce the state space rather than provide novel ways of reasoning. Another approach related to ours is [2].

tools, there are several points that need to be addressed: allow more generous unfolding and polyvariance (efficiency, both of the specialisation process and the specialised program, are less of an issue in model checking) to enable more precise residual programs and implement the full algorithm of [11] which allows for more fine grained abstraction and use BDD-like representations whenever possible. Currently we can only verify safety properties (i.e., no bad things happen) and not liveness properties (i.e., good things will eventually happen). The latter might be feasible by a more sophisticated support for the negation.

References

1. S.M. Abramov, R. Glück. Semantics modifiers: An approach to non-standard semantics of programming languages. In M. Sato, Y. Toyama (eds.), *International Symposium on Functional and Logic Programming*, 247–270. World Scientific, 1998.
2. M. Bruynooghe, H. Vandecasteele, A. de Waal. Detecting Unsolvable Queries for Definite Logic Programs. In C. Palamidessi, H. Glaser, K. Meinke (eds.) *Principles of Declarative Programming*, LNCS 1490, 118–133. Springer-Verlag, 1998.
3. R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
4. E.M. Clarke, E.A. Emerson, A. Sistla. Automatic verification of finite-state concurrent systems using temp. logic specifications. *ACM TOPLAS*, 8(2):244–263, 1986.
5. E.M. Clarke, J.M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, 1996.
6. C. Consel, S.C. Khoo. Parameterized partial evaluation. *ACM TOPLAS*, 15(3):463–493, 1993.
7. R. Glück. On the generation of specializers. *Journal of Functional Programming*, 4(4):499–514, 1994.
8. P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, B. Walters. Questions and answers about ten formal methods. In S. Gnesi, D. Latella (eds.) *Formal Methods for Industrial Critical Systems*, pages 179–203. Trento, Italy, 1999.
9. J. Hatcliff, M. Dwyer, S. Laubach. Staging analysis using abstraction-based program specialization. In C. Palamidessi, H. Glaser, K. Meinke (eds.) *Principles of Declarative Programming*, LNCS 1490, 134–151. Springer-Verlag, 1998.
10. N.D. Jones. The essence of program transformation by partial evaluation and driving. In N.D. Jones, M. Hagiya, M. Sato (eds.) *Logic, Language and Computation*, LNCS 792, 206–224. Springer-Verlag, 1994.
11. M. Leuschel. Program specialisation and abstract interpretation reconciled. In J. Jaffar (ed.) *JICSLP'98*, 220–234. MIT Press, 1998.
12. M. Leuschel, D. De Schreye. Logic program specialisation: How to be more specific. In H. Kuchen, S. Swierstra (eds.) *Programming Languages: Implementations, Logics and Programs.*, LNCS 1140, 137–151. Springer-Verlag, 1996.
13. M. Leuschel, B. Martens, D. De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM TOPLAS*, 20(1):208–258, 1998.
14. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, D.S. Warren. Efficient model checking using tabled resolution. In O. Grumberg (ed.) *Computer-Aided Verification*, LNCS 1254, 143–154. Springer-Verlag, 1997.
15. B.J. Ross. Running programs backwards: The logical inversion of imperative computation. *Formal Aspects of Computing*, 9:331–348, 1997.

16. K. Sagonas, T. Swift, D.S. Warren. XSB as an efficient deductive database engine. In *Intern. Conference on the Management of Data*, 442–453. ACM Press, 1994.
17. B. Steffen (ed.). *Tools and Algorithms for the Construction and Analysis of Systems*, LNCS 1384. Springer-Verlag, 1998.
18. V.F. Turchin. Program transformation with metasystem transitions. *Journal of Functional Programming*, 3(3):283–313, 1993.