Extending Explanation-Based Generalization by Abstraction Operators[*]

Igor Mozetič
Austrian Research Institute for Artificial Intelligence
Schottengasse 3, A-1010 Vienna, Austria
igor@ai-vie.uucp

Christian Holzbaur
Austrian Research Institute for Artificial Intelligence, and
Department of Medical Cybernetics and Artificial Intelligence
University of Vienna
Freyung 6, A-1010 Vienna, Austria
christian@ai-vie.uucp

Abstract

We present two contributions to the explanation-based generalization techniques. First, the operationality criterion is extended by abstraction operators. These allow for the goal concept to be reformulated not only in terms of operational predicates, but also allow to delete irrelevant arguments, and to collapse indistinguishable constants. The abstraction algorithm is presented and illustrated by an example. Second, the domain theory is not restricted to variables with finite (discrete) domains, but can deal with infinite (e.g., real-valued) domains as well. The interpretation

[*]Appears in *Machine Learning — EWSL-91* (Y. Kodratoff, Ed.), Proc. *European Working Session on Learning*, pp. 282-297, Porto, Portugal, March 6-8, 1991, Springer-Verlag (LNAI 482).

and abstraction are effectively handled through constraint logic programming mechanisms. In the paper we concentrate on the role of CLP($\Re$) — a solver for systems of linear equations and inequalities over reals.

1 Introduction

Explanation-based generalization (EBG) is a technique to formulate general concepts on the basis of an individual example and a domain theory (Mitchell *et al.* 1986). The domain theory is used to explain the training example, and the resulting explanation is generalized and reformulated in operational terms. Specifically, the input to the EBG algorithm consists of:

- Domain theory,
- Goal concept — concept to be learned,
- Training example — an instance of the goal concept,
- Operationality criterion — easily evaluable predicates from the domain theory in terms of which the goal concept should be reformulated.

It has been shown (Van Harmelen & Bundy 1988) that in the context of logic programming, EBG is essentially equivalent to partial evaluation. The domain theory is represented by a logic program, and the explanation is a proof that the example logically follows from the program. The goal concept is the partially evaluated program, i.e., it is reformulated in terms of leaves of the proof tree which contain only operational predicates. Further, the training example can be either ground, partially instantiated, or even omitted, since its role is just to restrict the search for the proof. In the case that no example is provided, the resulting goal concept is equivalent to the original one, but expressed just in terms of operational predicates and therefore more efficient for subsequent use.

In the paper we push further the idea of performing EBG by partial evaluation. In EBG the operationality criterion refers just to predicates — procedure calls of non-operational predicates are unfolded. We extend the criterion to abstraction operators which refer to all parts of atomic formulae. For example, predicates can become operational by deleting some arguments, or different constants can be collapsed into singletons. More systematically, if P is a domain theory represented by a logic program then it can be abstracted into P' in the following ways:

- by unfolding non-operational predicates throughout P (like in EBG),
- by deleting some arguments of functions and predicates throughout P,

In contrast to partial evaluation (without training example), the abstracted goal concept and domain theory are no longer equivalent to the original ones, but are their logical consequences, respectively (Figure 1). The equivalence is lost due to the arguments deletion and many-to-one renaming. As a consequence, a non-theorem from the detailed space can be abstracted into a theorem in the abstract space. Each theorem is abstracted into a theorem, however. A specific type of abstractions with this property are so-called *truthful* or *TI-abstractions* (theorem increasing, Giunchiglia & Walsh 1989). In general, an abstraction is a *partial* and not a total mapping (Mozetic 1990a) since one wants to ignore irrelevant features of the domain theory for the task at hand. Note also that in contrast to EBG and partial evaluation, an abstraction might introduce a different language into the abstract space.

The abstracted domain theory can be used either to answer the queries more efficiently and less precisely, or in conjunction with the original domain theory as a falsity preserving filter. Abstractions turned out to be useful in reducing the search space in theorem proving (Plaisted 1981, Giunchiglia & Walsh 1990), planning (Sacerdoti 1974, Korf 1987), and model-based diagnosis (Gallanti *et al.* 1989, Mozetic 1990b, Mozetic & Holzbaur 1991).

Section 2 motivates the introduction of abstraction operators by an example. Given is a domain theory for a numerical model of an inverter. In section 3 the three abstraction operators which replace the operationality criterion are defined. The numerical model is first abstracted to a qualitative model, and then to a binary-logic model of an inverter. Section 4 describes the abstraction algorithm and its Prolog implementation. In section 5 constraint logic programming inference over real-valued variables is outlined and its application is illustrated by the inverter example.

2 Motivating example: a model of an inverter

We represent a domain theory by a constraint logic program (CLP, Jaffar & Michaylov 1987) which is a logic program extended by interpreted functions. A proper implementation of a CLP scheme allows for an easy integration of specialized problem solvers into the logic programming framework. For example, in Metaprolog (an extension of C-Prolog, Holzbaur 1990) specialized solvers communicate with the standard Prolog interpreter via extended

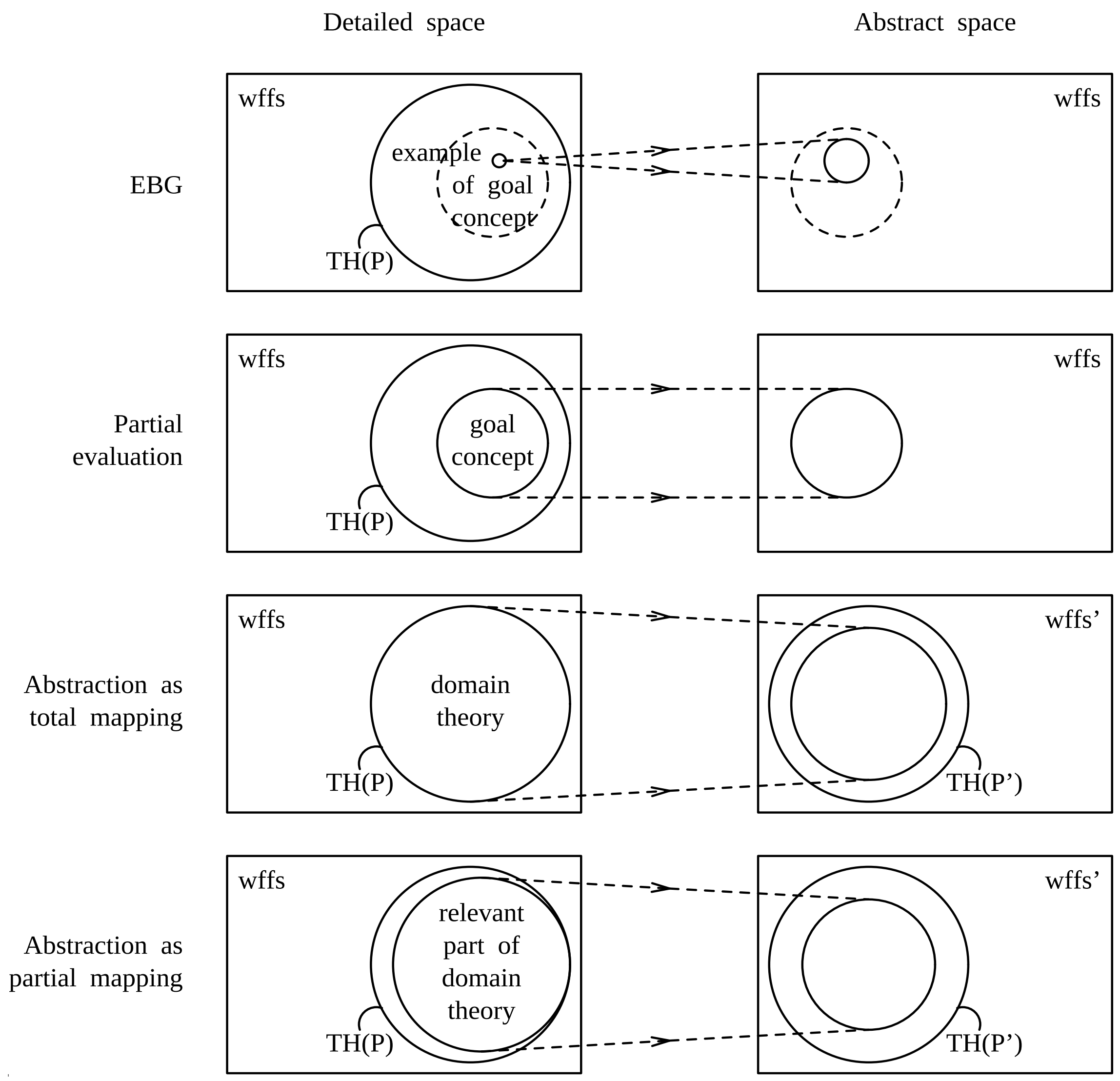

Figure 1: Relation between EBG, partial evaluation, and abstractions. P and P' denote a detailed and an abstract domain theory, respectively, and *TH(P)* is the set of well formed formulae provable from P.

- by renaming constant, function, and predicate symbols throughout P (the renaming is typically many-to-one).

semantic unification and are implemented in Prolog themselves. So far, three solvers have been implemented: constraint propagation over finite domains by forward checking, CLP($\mathcal{B}$) — a solver over boolean expressions, and CLP($\Re$) — a solver for systems of linear equations and inequalities over reals (Holzbaur 1990). In the paper we concentrate on the role of CLP($\Re$) in the interpretation and abstraction of a domain theory.

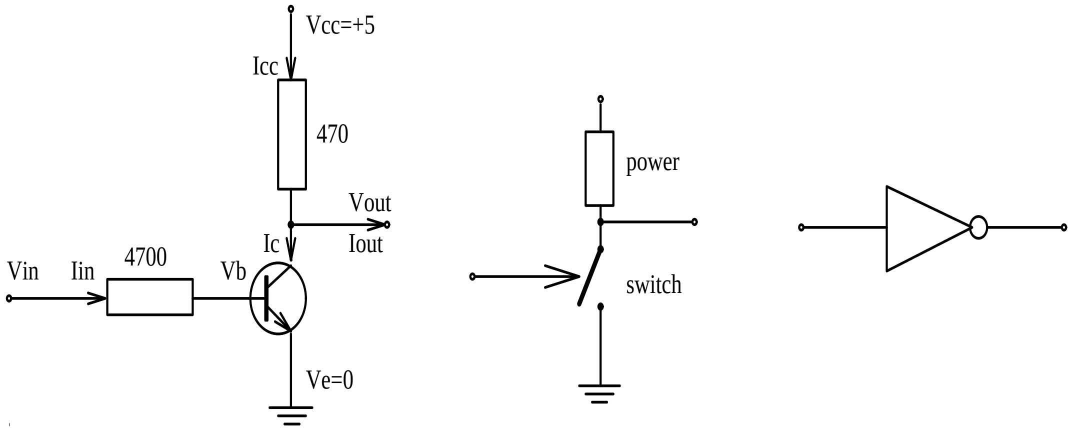

Figure 2: A numerical, qualitative, and logical model of an inverter.

We illustrate the EBG algorithm, realized by partial evaluation and without a training example, on a model of an inverter. The initial domain theory consists of a numerical model of an inverter (predicate inv_n), which is later simplified into a qualitative (inv_q), and finally into a logical model (inv_l, Figure 2). The inverter is realized by an npn transistor and two resistors. The description of an npn transistor is from Heinze, Michaylov & Stuckey (1987). The transistor operates in three states: *cutoff*, *saturated*, and *active*. In digital circuits only the cutoff and saturated states are of interest, and therefore the active state, interesting in amplifier circuits, is omitted. *Vx* and *Ix* denote the real-valued voltages and currents for the base, collector and emmiter, respectively. Constants *Beta*, *Vbe*, and *Vcesat* are device parameters.

Domain theory:

```
inv_n( S, b(Vin,Iin), b(Vout,Iout) )  ←
        switch_n( S, Vin, Iin, Vout, Ic ),
        power_n( Ic, Vout, Iout ).

switch_n( S, Vin, Iin, Vc, Ic )  ←
        Ve=0, Beta=100, Vbe=0.7, Vcesat=0.3,
        resistor( Vin, Vb, Iin, 4700 ),
        transistor( S, Beta, Vbe, Vcesat, Vb, Vc, Ve, Iin, Ic, Ie ).
```

$power_n(Ic, Vout, Iout) \leftarrow$
$\quad Vcc{=}5,\ Ic{+}Iout{=}Icc,$
$\quad resistor(Vcc, Vout, Icc, 470),$
$\quad 0{\leq}Iout,\ Iout{\leq}0.006.$

$resistor(V1, V2, I, R) \leftarrow R{>}0,\ V1{-}V2{=}I{*}R.$

$transistor(cutoff, Beta, Vbe, Vcesat, Vb, Vc, Ve, Ib, Ic, Ie) \leftarrow$
$\quad Vb{<}Ve{+}Vbe,\ Ib{=}0,\ Ic{=}0,\ Ie{=}0.$
$transistor(saturated, Beta, Vbe, Vcesat, Vb, Vc, Ve, Ib, Ic, Ie) \leftarrow$
$\quad Vb{=}Ve{+}Vbe,\ Vc{=}Ve{+}Vcesat,\ Ib{\geq}0,\ Ic{\geq}0,\ Ie{=}Ic{+}Ib.$

Goal concept: $inv_n(State, In, Out)$.

Operational predicates: $=, <, \leq, >, \geq$.

Partial evaluation of the goal concept unfolds procedure calls of non-operational predicates, propagates constant values, and branches out conditionals. Using Metaprolog with CLP($\Re$), this yields the following operational definition of the inverter.

Reformulated goal concept (through partial evaluation):

$inv_n(cutoff, b(Vin,Iin), b(Vout,Iout)) \leftarrow$
$\quad Vin{<}0.7,$
$\quad Iin{=}0,$
$\quad Vout{=}{-}470{*}Iout{+}5,$
$\quad Iout{\leq}0.006,\ Iout{\geq}0.$
$inv_n(saturated, b(Vin,Iin), b(Vout,Iout)) \leftarrow$
$\quad Vin{=}4700{*}Iin{+}0.7,$
$\quad Iin{\geq}0,$
$\quad Vout{=}0.3,$
$\quad Iout{\leq}0.006,\ Iout{\geq}0.$

The reformulated definition of the inverter is structureless and refers just to easily evaluable arithmetic operators. However, it is still unnecessarily detailed for a number of tasks. For diagnosis, for example, it does not really matter if the voltage is 4.4 or 4.6, but whether it is qualitatively *high* or *low*, and whether the transistor properly operates as a switching device. If we specify the transistor state and currents as irrelevant arguments, and voltages 0–0.7 and 2–5 as indistinguishable, we can derive the following alternative reformulation of the domain theory.

Alternative reformulation (through abstractions):

inv_l(low, high).
inv_l(high, low).

In the following section we specify abstraction operators more precisely. We illustrate their applicability by showing a two-step abstraction: a qualitative model of the inverter is derived first, and then abstracted into the above logical model.

3 Abstraction operators

Underlying the formulation of abstractions is a typed logic program (Lloyd 1987). Types provide a natural way of expressing the concept of a domain and are convenient for specifying abstraction operators in a compact form. We assume that variables and constants have types such as τ. Functions have types of the form $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$, and predicates have types of the form $\tau_1 \times \ldots \times \tau_n$.

The following three abstraction operators replace and extend the EBG operationality criterion. Since they do not change the structure of formulae but refer just to atoms they define a class of *atomic abstractions* (Giunchiglia & Walsh 1990). We use a binary predicate h_τ to denote abstractions of constants and functions of range type τ, and a binary predicate h to denote predicate abstractions.

1. Collapsing constants.
   Different constants can be renamed into a single constant. For example, assume that a_1 and a_2 are of type τ, and that they are collapsed into a single constant a':
   $$h_\tau(a_1, a'). \qquad h_\tau(a_2, a').$$
2. Function abstractions.
   Functions can be renamed and irrelevant arguments deleted. For example, let f be of type $\tau_1 \times \ldots \times \tau_n \rightarrow \tau$, its first argument be deleted, and f be renamed to f':
   $$h_\tau(f(X_1, X_2 \ldots, X_n), f'(X'_2, \ldots, X'_n)) \leftarrow h_{2\tau}(X_2, X'_2), \ldots, h_{n\tau}(X_n, X'_n).$$
3. Predicate abstractions.
   Operational predicates can be renamed and some arguments deleted. For example, let p be of type $\tau_1 \times \ldots \times \tau_n$, its first argument be deleted, and p be renamed to p':
   $$h(p(X_1, X_2 \ldots, X_n), p'(X'_2, \ldots, X'_n)) \leftarrow h_{2\tau}(X_2, X'_2), \ldots, h_{n\tau}(X_n, X'_n).$$

The abstraction operators degenerate to the EBG operationality criterion as a special case. In EBG one specifies just the predicate abstractions by adding, for each operational predicate p, a unit clause of the form:

$$h(p(X_1, \ldots, X_n), p(X_1, \ldots, X_n)).$$

Predicates for which no abstraction is specified are assumed to be non-operational and their definitions are unfolded.

The following specifies the extended operationality criterion for the inverter example.

Collapsing constants:

$h_s(cutoff,\ ok).$
$h_s(saturated,\ ok).$

$h_i(0,\ zero).$
$h_i(I,\ pos) \ \leftarrow\ I{>}0.$ *% negative I has no abstraction*

$h_v(V,\ low) \ \leftarrow\ 0{\leq}V,\ V{<}0.7.$
$h_v(V,\ high) \ \leftarrow\ 2{\leq}V,\ V{\leq}5.$

Function abstraction:

$h_b(b(V,\_I),\ V') \ \leftarrow\ h_v(V,V').$ *% I is deleted*

Predicate abstractions:

$h(inv_n(S,X,Y),\ inv_q(S',X',Y')) \ \leftarrow$
$\quad h_s(S,S'),\ h_b(X,X'),\ h_b(Y,Y').$

$h(switch_n(S,Vin,\_Iin,Vc,Ic),\ switch_q(S',Vin',Vc',Ic')) \ \leftarrow$ *% Iin is deleted*
$\quad h_s(S,S'),\ h_v(Vin,Vin'),\ h_v(Vc,Vc'),\ h_i(Ic,Ic').$

$h(power_n(Ic,Vout,\_Iout),\ power_q(Ic',Vout')) \ \leftarrow$ *% Iout is deleted*
$\quad h_i(Ic,Ic'),\ h_v(Vout,Vout').$

From the numerical model of the inverter and the above abstractions, a qualitative model was automatically derived through term rewriting and partial evaluation. Predicates, for which no abstractions are specified (*resistor, transistor*) are treated as non-operational and their definitions are unfolded. The remaining predicates and terms are rewritten according to the abstraction specifications.

Reformulated goal concept (qualitative model of the inverter):

$inv_q(\ S,\ Vin,\ Vout\ ) \ \leftarrow$
$\quad switch_q(\ S,\ Vin,\ Vout,\ Ic\ ),$
$\quad power_q(\ Ic,\ Vout\ ).$

$switch_q$(ok, low, _, zero).
$switch_q$(ok, high, low, zero).
$switch_q$(ok, high, low, pos).

$power_q$(zero, high).
$power_q$(pos, low).
$power_q$(pos, high).

In the next step $switch_q$ and $power_q$ are treated as non-operational, inv_q is renamed to inv_l, and the argument S which denotes the internal state of the inverter is deleted.

Predicate abstraction:

h(inv_q(_S,X,Y), inv_l(X,Y)). *% S is deleted*

The abstraction algorithm yields the binary-logic model of the inverter.

Reformulated goal concept (logical model of the inverter):

inv_l(low, high).
inv_l(high, low).

4 The abstraction algorithm

The algorithm takes as an input a goal concept, a domain theory, and abstraction operators. The goal concept is an explicit parameter to the algorithm, while the domain theory and abstraction operators are implicit inputs, accessible through the Prolog built-in *clause/2* and *call/1* predicates. A training example can be represented as a partially instantiated goal concept, e.g., *inv_n(S,b(0.5,Iin),b(Vout,0.001))*. However, we do not address the advantages and disadvantages of providing an example here — the algorithm works in both cases.

The output of the algorithm is an abstracted goal concept in the form of a predicate definition, i.e., a set of clauses with the same head. The procedure is run in a failure-driven loop. At each iteration a clause defining the goal concept is selected, abstracted, and written to the output:

```
abstract_goal( Goal )  ←
        abstract_clause( Goal, Clause' ),
        write_clause( Clause' ),
        fail.
```

```
abstract_goal( Goal ).
```

The procedure *write_clause(Clause')* extracts residual constraints over relevant variables, adds them to the body of the abstracted clause, and outputs the clause (see next section). A clause is abstracted by first unfolding definitions of non-operational predicates, abstracting the remaining atoms in the body, and finally abstracting the head of the clause. For brevity, we omit recursive definitions of predicates which traverse a list of atoms (*unfold_atoms, abstract_atoms*) and present just the base cases (*unfold_atom, abstract_atom*):

```
abstract_clause( Head, (Head' ← Body') )  ←
        clause( Head, Body ),
        unfold_atoms( Body, Leaves ),
        abstract_atoms( Leaves, Body' ),
        abstract_atom( Head, Head' ).

unfold_atom( Atom, Leaves )  ←
        not clause( h(Atom, _), _ ), !,     % non-operational predicate
        unfold( Atom, Leaves ).
unfold_atom( Atom, Atom ).

unfold( Atom, true )  ←  builtin( Atom ), !,     % evaluate built-ins
        call( Atom ).
unfold( Atom, Leaves )  ←
        clause( Atom, Body ),
        unfold_atoms( Body, Leaves ).

abstract_atom( Atom, Atom' )  ←     % predicate abstractions
        clause( h(Atom, Atom'), Arguments ),
        abstract_terms( Arguments ).
```

A predicate is abstracted by referring to the abstraction operator h which defines the new name and arity, and then by abstracting the arguments. A term abstraction depends on its type: a variable is 'abstracted' to the same variable, a constant of type τ is renamed according to the h_τ abstraction operator, and a function of range type τ is abstracted by referring to the abstraction operator h_τ and by recursively abstracting its arguments:

```
abstract_term( hτ(Variable, Variable) )  ←  var( Variable ), !.
abstract_term( hτ(Constant, Constant') )  ←  constant( Constant ), !,
        call( hτ(Constant, Constant') ).
abstract_term( hτ(Term, Term') )  ←     % structured term
```

clause(h_τ(Term, Term'), Arguments),
abstract_terms(Arguments).

constant(X) $\leftarrow$ *atomic(X).* *% constant or number*
constant(X) $\leftarrow$ *ismeta(X).* *% constrained variable in Metaprolog*

Note that a variable is treated as a constant if it is constrained and the constraints are satisfiable, i.e., if it can be substituted by a constant. See the next section for an example.

The following example illustrates the algorithm for a simple case when just predicates and functions are renamed and some arguments deleted. A standard Prolog interpreter suffices for such abstractions. The next section presents a more involved example where a Prolog interpreter has to be augmented by the CLP($\Re$) solver in order to collect linear constraints over reals and verify their satisfiability.

Example. Assume that the following clause is to be abstracted:

inv_n(S, b(Vin,Iin), b(Vout,Iout)) $\leftarrow$
$switch_n$(S, Vin, Iin, Vout, Ic),
$power_n$(Ic, Vout, Iout).

Both atoms in the body are operational and no unfolding takes place. *$Switch_n$* is abstracted to *$switch_q$* with the argument *Iin* deleted, and *$power_n$* is abstracted to *$power_q$* with the argument *Iout* deleted:

inv_n(S', b(Vin',Iin), b(Vout',Iout)) $\leftarrow$
$switch_q$(S', Vin', Vout', Ic'),
$power_q$(Ic', Vout').

The head of the clause *inv_n* is abstracted to *inv_q* by abstracting its arguments. The variable *S'* remains a variable, and both terms *b(V,I)* are abstracted according to the function abstraction specified by the h_b clause. As a result, currents *I* are dropped and only voltages *V'* remain:

inv_q(S', Vin', Vout') $\leftarrow$
$switch_q$(S', Vin', Vout', Ic'),
$power_q$(Ic', Vout').

5 The role of CLP($\Re$) in abstractions

Whereas the previous abstraction of the inv_n clause did not rely on any CLP($\Re$) functionality, the abstraction of $switch_n$ to $switch_q$ does, however. All atoms in the body of $switch_n$ are non-operational so that *unfold_atoms* in the abstraction algorithm essentially calls $switch_n(S, Vin, Iin, Vc, Ic)$ and returns the following two answer substitutions with residual constraints:

$switch_n($ *cutoff, Vin, 0, Vc, 0* $)$
  Constraints: $Vin<0.7$;

$switch_n($ *saturated, Vin, Iin, 0.3, Ic* $)$
  Constraints: $Vin=4700*Iin+0.7,\ Vin\geq 0.7,\ Ic\geq 0$

Abstraction of the first solution. The abstraction algorithm proceeds with the abstraction of the head of $switch_n$ for there are no body goals left. The constant *cutoff* is abstracted into *ok* through an application of h_s. The next argument *Vin* is a constrained variable and there are two h_v clauses that specify the abstraction rules for voltages. The first clause succeeds as the combined set of constraints $\{Vin<0.7,\ 0\leq Vin,\ Vin<0.7\}$ is satisfiable. *Vin* is abstracted to *low*. The argument *Iin* is deleted through predicate abstraction. The fourth argument *Vc* is again a voltage. It remains unchanged in the abstraction since it is unconstrained. The last argument *Ic* is instantiated to 0 which is abstracted to *zero* in turn. This yields the first defining clause of the abstract $switch_q$ predicate:

$switch_q($ *ok, low, Vc, zero* $)$.

During backtracking the abstraction of *Ic* to *pos* fails as this is incompatible with the constraint $Ic>0$. Backtracking proceeds until *transistor* and therefore $switch_n$ delivers the second solution.

Abstraction of the second solution. Again the abstraction algorithm proceeds with the head of $switch_n$ for there are no body goals left. The constant *saturated* is abstracted into *ok* through an application of h_s. The next argument *Vin* is a constrained variable and there are two h_v clauses that specify the abstraction rules for voltages:

$h_v(V,\ low)\ \leftarrow\ 0\leq V,\ V<0.7.$
$h_v(V,\ high)\ \leftarrow\ 2\leq V,\ V\leq 5.$

The first clause fails as the combined set of constraints $\{Vin\geq 0.7,\ 0\leq Vin,\ Vin<0.7\}$ is unsatisfiable. The second clause, however, succeeds with the abstraction of *Vin* to *high*.

The argument *Iin* is deleted through predicate abstraction. The fourth argument *Vc*, a voltage, is instantiated to 0.3. This value satisfies the constraints of the first clause of h_v and *Vc* is abstracted to *low*. The last argument *Ic* is a constrained variable and there are two h_i clauses that specify the abstraction rules for currents:

$h_i(0,\ zero).$
$h_i(I,\ pos)\ \leftarrow\ I{>}0.$

The residual constraint $Ic{\geq}0$ from $switch_n$ is compatible with both clauses of h_i, and therefore we get the final two defining clauses of the abstract $switch_q$ predicate:

$switch_q(\ ok,\ high,\ low,\ zero\ ).$
$switch_q(\ ok,\ high,\ low,\ pos\ ).$

The role of CLP($\Re$) in this derivation is manyfold. First, it allows to compute with a domain theory over real-valued variables at the numerical level of the model. Second, it admits the specification of abstraction operators that share the computational domain with the numerical model. The third function is to guarantee the satisfiablity of the constraints collected during partial evaluation and abstraction. The ability to deal with partially instantiated constraints is vital if we want to apply EBG in the *absence* of an example — in particular if we operate in an infinite domain.

One traditional method for deciding linear inequalities is the simplex method. The simplex method works by turning inequalities into equations through the introduction of so-called 'slack variables'. This leads to a 'contamination' of the equation system with artificial variables from the user's point of view. In our experience, the amount of code that is needed to compute a human readable form of the (in)equation system is unproportionally high in comparison to the code that does the actual job. Therefore, we rather selected the Shostak's 'Loop Residue' method (Shostak 1981). Besides being better suited for small inequalities, this method operates with a 'direct' representation of inequalities. For each set of inequalities a graph is constructed whose vertices correspond to the variables and edges to the inequalities. Kraemer (1989) proves the equivalence between a satisfiable set of inequalities and the corresponding closed graph without any infeasible loop.

As far as the derivation of the abstract model is concerned, the residual set of constraints that are *not* related to the variables in the abstract model are not used any longer. They *can* be added to the numerical model, however. This leads to a specialization of the model or a *relevance projection* with respect to the abstractions. In analogy to the traditional EBG framework we can thereby prune irrelevant computations when the *numerical* model is used for diagnosis, say.

Our Metaprolog implementation of CLP($\Re$) is preferred over other existing implementations of CLP($\Re$) (Heintze *et al.* 1987, Jaffar 1990) since it allows for the simultaneous use of solvers for different domains in a consistent framework. In this respect Metaprolog is very well suited for the computational demands that arise in the context of hierarchical abstractions. The numerical level of the model can be formulated with CLP($\Re$) for example, and successive qualitative abstractions thereof typically utilize constraint propagation over finite domains. The implementation of the specialized solvers is based on the user-definable[1] extended unification. As the solvers themselves are written in Prolog they can easily be customized to specific demands. The choice of Prolog as an implementation language for the equation solver leads to a reduction in code size by an order of magnitude. On the other hand, the CLP($\Re$) solver implemented in *interpreted* Prolog is about six times slower than a dedicated C implementation (Holzbaur 1990). More important, however, is the fact that the solvers operate on the same data structures — Prolog terms in fact. Therefore it is easy to combine different, independent solvers in the same program. Additionally, the user is supplied with an external representation of the residual constraints which can be manipulated further (printed, for example).

6 Conclusion

The paper makes two contributions to the EBG techniques. First, the proposed scheme is less dependent on a training example. We can even get away without an example and still perform nontrivial and useful computations at the partial evaluation time. If we accept that one intention behind the EBG is to derive specialized, and therefore more efficient reformulations of the domain theory, it is obvious that the choice of a richer computational domain contributes to this aspect. If the examples of this paper were to be partially evaluated in an empty semantic theory (pure Prolog with the Herbrand interpretation) one would have to declare all relational operators and equations over real-valued variables of the domain theory as operational. This results in specializations where the set of collected constraints (operational predicates) is neither minimal, nor is it guaranteed to be satisfiable. The choice of a richer computational domain allows for the resolution of some constraints at the EBG time already.

The second contribution are abstraction operators which extend the standard EBG operationality criterion. Further, abstractions perform a kind of 'relevance projection' of the

[1]In Metaprolog, CLP($\Re$), CLP($\mathcal{B}$), and forward checking over finite domains are provided as libraries.

domain theory even in the absence of an example. Through this explicit specification, the usual problem of estimating the relevance of training examples is circumvented. In the case of a domain theory over infinite domains, it is only the combination of abstractions with a powerful computational domain that allows for the application (a partial or complete resolution) of the abstraction operators at the EBG time.

Acknowledgements

This work was supported by the Austrian Federal Ministry of Science and Research. Thanks to Bernhard Pfahringer for valuable comments and to Robert Trappl for making this work possible.

References

Gallanti, M., Roncato, M., Stefanini, A., Tornielli, G. (1989). A diagnostic algorithm based on models at different level of abstraction. *Proc. 11th IJCAI*, pp. 1350-1355, Detroit, Morgan Kaufmann.

Giunchiglia, F., Walsh, T. (1989). Abstract theorem proving. *Proc. 11th IJCAI*, pp. 372-377, Detroit, Morgan Kaufmann.

Giunchiglia, F., Walsh, T. (1990). Abstract theorem proving: mapping back. IRST Technical Report 8911-16, Istituto Ricerca Scientifica e Tecnologica, Trento, Italy.

Heintze, N., Jaffar, J., Michaylov, S., Stuckey, P., Yap, R. (1987). The CLP($\Re$) programmer's manual. Dept. of Computer Science, Monash University, Australia.

Heintze, N., Michaylov, S., Stuckey, P. (1987). CLP($\Re$) and some electrical engineering problems. *Proc. 4th Intl. Conference on Logic Programming*, pp. 675-703, Melbourne, Australia, The MIT Press.

Holzbaur, C. (1990). Specification of constraint based inference mechanisms through extended unification. Ph.D. Thesis, Technical University of Vienna, Austria.

Jaffar, J. (1990). CLP($\Re$) version 1.0 reference manual. IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY.

Jaffar, J., Michaylov, S. (1987). Methodology and implementation of a CLP system. *Proc. 4th Intl. Conference on Logic Programming*, pp. 196-218, Melbourne, Australia, The MIT Press.

Korf, R.E. (1987). Planning as search: a quantitative approach. *Artificial Intelligence 33*, pp. 65-88.

Kraemer, F.-J. (1989). A decision procedure for Presburger arithmetic with functions and equality. SEKI working paper SWP-89-4, FB Informatik, University of Kaiserslautern, Germany.

Lloyd, J.W. (1987). *Foundations of Logic Programming* (Second edition). Springer-Verlag, Berlin.

Mitchell, T., Keller, R., Kedar-Cabelli, S. (1986). Explanation-based generalization: A unifying view. *Machine Learning 1 (1)*, pp. 47-80.

Mozetic, I. (1990a). Abstractions in model-based diagnosis. Report TR-90-4, Austrian Research Institute for Artificial Intelligence, Vienna, Austria. *Proc. Automatic Generation of Approximations and Abstractions, AAAI-90 Workshop*, pp. 64-75, Boston.

Mozetic, I. (1990b). Reduction of diagnostic complexity through model abstractions. Report TR-90-10, Austrian Research Institute for Artificial Intelligence, Vienna, Austria. *Proc. First Intl. Workshop on Principles of Diagnosis*, pp. 102-111, Stanford University, Palo Alto.

Mozetic, I., Holzbaur, C. (1991). Integrating qualitative and numerical models within Constraint Logic Programming. Report TR-91-2, Austrian Research Institute for Artificial Intelligence, Vienna, Austria. *Workshop on Qualitative Reasoning about Physical Systems*, Genova, Italy.

Plaisted, D.A. (1981). Theorem proving with abstractions. *Artificial Intelligence 16*, pp. 47-108.

Sacerdoti, E.D. (1974). Planning in a hierarchy of abstraction spaces. *Artificial Intelligence 5*, pp. 115-135.

Shostak, R. (1981). Deciding linear inequalities by computing loop residues. *Journal of the ACM 28 (4)*, pp. 769-779.

Van Harmelen, F., Bundy, A. (1988). Explanation-based generalisation = partial evaluation. *Artificial Intelligence 36*, pp. 401-412.