PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Theorem Proving Engine and Strategy Description Language

Massimo Bruschi

State University of Milan - Computer Science Department
Via Comelico 39, 20135 Milan, Italy
e-mail: mbruschi@imiucca.csi.unimi.it

Abstract

The concepts of strategy description language (*SDL*) and theorem proving engine (*TPE*) are introduced as architectural and applicative tools in the design and use of an automated theorem proving system. Particular emphasis is given to the use of an SDL as a research tool as well as a way to use a prover both as a batch or as an interactive program. In fact, the availability of an interpreter for such a language offers the possibility of having a system able to cover both of these usages, giving to the user some way of choosing the granularity of the steps the prover must take. Three examples are given to show possible applications. Their purpose is to show its usefulness for expressing and testing new ideas. Some interesting capabilities of an SDL are applied to highlight how it allows the treatment of self-analysis on the state of the search space. Examples of these are the definition of a self-adaptive search and a tree pruning strategy. All the definitions we give reflect a running Prolog prototype and inherit much from the Prolog style and structure.

1 Introduction

The uses of and the interest in automated theorem proving have grown markedly in the preceding decade. The cause rests in part with faster computers, easy access to workstations, portable and powerful automated theorem-proving programs, and successes with answering open questions. Various researchers in the field conjecture that far more power is needed to attack the deep problems of mathematics and logic that are currently out of reach.

Although some of the needed increase in effectiveness will result from even faster computers, many state that the real advances will result from the formulation of new and diverse strategies. Because we feel that the ease of comparing, analyzing, and formulating such strategies would be enhanced if an appropriate abstract language and theory were available, we undertake here the development of such a language. Perhaps the abstraction and language will lead to needed insights into the nature of strategy of diverse types. In addition, because of its relation to this language, here we also provide an abstract treatment of theorem-proving programs as engines. This abstraction may enable researchers to analyze the differences, similarities, and sources of power among the radically diverse program designs.

The idea for developing a strategy description language (SDL) usable to define search strategies for a theorem prover was born when we began to study the application of parallelism to ATP. One proposal was to run many theorem provers on the same problem but with different search strategies. Having different strategies expressed as programs would mean having, as input of each prover process, the couple $< theorem, search - algorithm >$.

The development of a language requires the definition of an abstract machine to execute its programs, requiring an interpreter for the language. Our experiences and previous work with Prolog has suggested its use for the realization of a prototype.

One simple way to build an interpreter is to define a kernel module offering the basic services. This led us to the definition of a theorem-proving engine (TPE). Next, we developed a theorem prover having an SDL interpreter and a TPE as basic modules. *zSDL* is the name of our SDL.

Generally, we conjecture that an SDL might benefit by having one (or more) of the basic attitudes and of being procedural, functional, and logical. It should also be able to focus on the operations with different granularity as well as directing the prover process, controlling details of different level of complexity. As a sample model we can think at production systems in AI, and say that an SDL could be used to describe the control side of such a system. There can be as many SDL languages as different production systems.

The language we defined did not result from a deep analysis of the cited aspects; instead, it has been driven by the underlying structure of the TPE we developed, by the fact that is realized in Prolog, and by the wish to define the language *on the field* so that it could be run. One of the nice things about Prolog is that you can develop executable meta-languages.

2 A theorem proving engine

A TPE is a program module devoted to maintain and operate a knowledge base *(KB)* of logical formulas and a set of indexes on them. We think of these indexes as sets of references (or ids) to the formulas. The sets are distinguished by name. Each formula is retained together with various information about it.

A TPE can perform two basic activities: *inference* and *reduction*. The object of the first activity is to deduce new knowledge, gathering it by considering various subsets of the formulas in the KB. The object of the second activity is to keep the size (or the *weight*) of the KB as small as possible, by discarding redundant information. We require that every successful call to the inference process *(IP)* also calls the reduction process *(RP)*.

To better define the activities of a TPE, we focus on a possible minimal interface to such a module. We assume that the TPE finds the KB initialized with a given input set of formulas and that each operation maintains appropriately the indexes. We shall extend this interface gradually in the paper.

The kernel functions of a TPE can be:

- *(TPE.1)* - **enable**$(+Rule)$
- *(TPE.2)* - **disable**$(+Rule)$:

  A TPE is thought to offer a set of inference and reduction rules, each referred with a name. An IP will then apply the set of all the active inference rules, and the RP will only use the active reduction rules. These two functions are used to control the *activity sets*. For simplicity we assume the calls can also accept a list of rule names.

- *(TPE.3)* - **superpose**$(+Id_1, +Id_2)$:

  Its purpose is to activate the IP. It will superpose the formula referred to by Id_1 on the one referred to by Id_2 using all the active inference rules. We use the concept of superposition because it implies the ordering of the arguments, which is sometimes required. In this respect the general form of a single inference (as well as reduction) rule is thought of as

$$\frac{Id_1, Id_2}{\{N_1, N_2, \ldots, N_m\}}$$

  meaning that this rule takes as premises two formula references and produces a set of new references associated to the formulas resulting from the actual application. Consider for example the binary resolution inference rule. It takes two clauses and generates a set of resolvents. So, if we consider the clausal formulas referred to by Id_1 and by Id_2, the reference set $\{N_1, N_2, \ldots, N_m\}$ will refer to their resolvents (if any). Rules with single premises are called with **superpose**(Id, Id).

- *(TPE.4)* - **delete**$(+Id)$:

  It is used to delete the formula referred to by Id from the KB and from the indexes. This operation, combined with a superposition call, can be used to realize transformation processes on the KB. Consider for example the standard CNF transformation. It replaces a formula with a (satisfiability) equivalent set of clauses. We can model this by calling an inference rule with only one premise to generate the set of clauses and then delete the premise. As a matter of fact we think of this operation as reversible. See the next operation.

- *(TPE.5)* - **undelete**$(+Id)$:

  It is called to recover an earlier deletion of a formula. We can think of it as a special inference rule that uncovers a formula. It can be useful in adaptive searches. Suppose for example we are using a weighting strategy to discard newly generated formulas if they exceed a fixed weight. Using the **delete/1** call we can simply hide the formula from the KB and the indexes and later recover it if, for example, the search ends with a consistency status.

As a matter of fact the indexes on the formula KB have a dominant role for understanding the entire idea. In the next section we will make clearer this role.

3 zSDL: a strategy description language

Indexes, as sets of references to KB's formulas, are the basic objects of the language zSDL, which uses id-sets as the basic elements to refer to nodes and to describe the visit of the search tree.

The underlying idea is that an SDL requires some mechanism to represent a proof tree, for the ideal search strategy for proving a given theorem is the description of the precise steps the reasoning module must follow to reach the proof nodes in the search tree. With an SDL we must be able to speak about the nodes of the tree (the formulas) and the relations between them (how to reach the parents of each node, following the ancestor relation, as well as how to reach the children of a node, following the descendants relation). Another useful property might be the ability to know the level of a tree node, in order to define a (partial) ordering between the steps made to reach the proof (a sequence of parallelizable steps).

From these observations we chose to use sets of nodes as the basic description objects. And zSDL turned out to be, in some sense, a sets-operations oriented language. We will refer to a generic zSDL set of references to formulas to mean either an id-set or an index. A set is referred by a (unique) name. It is something like a variable of type id-set.

In zSDL we can apply to the id-sets all of the common operations and relations on sets, plus some special (procedural) ones like assignments, evaluation, etc. The following is a list of these functions, giving in addition some of the syntax of zSDL (recall that it is a Prolog by-product). In zSDL an id-set is represented as a Prolog list.

The Prolog variable names implicitly define the types of the operators in the following way:

SetName: the name of the variable that refers to the set.

SetExpr: an expression on sets, which can be an explicit set (list), a *SetName* or an expression built up using the defined operations.

Var: a Prolog non-instantiated variable.

ElemOrVar: a Prolog variable *(Var)* eventually instantiated *(Elem)*.

Notice that the *SetExpr* are evaluated.

◇ *(zSDL.1) - set operations* :

$+SetExpr_A$.# $+SetExpr_B$ % *union*
$+SetExpr_A$.+ $+SetExpr_B$ % *weak union*
$+SetExpr_A$.* $+SetExpr_B$ % *intersection*
$+SetExpr_A$.- $+SetExpr_B$ % *difference*

The weak union makes no checks on repetitions.

◇ *(zSDL.2) - set relations* :

$?ElemOrVar$.? $+SetExpr$ % *membership*
$+SetExpr_A$.=< $+SetExpr_B$ % *containment*
$+SetExpr_A$.< $+SetExpr_B$ % *strict containment*
$+SetExpr_A$.= $+SetExpr_B$ % *equality*

Notice that, using the Prolog negation, we also have the negations of these relations

◇ *(zSDL.3) - set procedures* :

$+SetName$:= $+SetExpr$ % *assignment*
$-Var$.? $SetName$ % *extract 1st element*
$-Var$ .. $+SetExpr$ % *evaluate*
:. $+SetName$ % *destroy the set*

The *pop* operation treats the set as a stack.

As an example, in a zSDL-Prolog session we could have:

```
| ?- a := [1,2,3],
     b := a .- [3,4,5].

yes
| ?- A .. a,
     B .. b,
     X .. b .# [6].

A = [1,2,3],
B = [1,2],
X = [1,2,6]
```

in which you see how zSDL sets are permanent objects, contrary to the classical Prolog variables.

This level of basic operations on (id-)sets must be enriched by statements to permit interaction with the TPE. We will show the basic calls zSDL defines to run an IP by developing the Prolog code that can realize it.

We are looking for a statement responsible for executing the actual inference steps applicable on some given id-sets. Consider the zSDL syntax

◇ *(zSDL.4) - directed superposition* :

$+SetExpr_A$ ++> $+SetExpr_B$

After the evaluation of the id-set expressions the general form of a call can be thought of as

$$[A_1, A_2, \ldots, A_n] \text{ ++> } [B_1, B_2, ..., B_m].$$

Obviously we expect this search to consider all the pairs, i.e. the TPE must be directed to try all the following superpositions:

$$\begin{array}{cccc} < A_1, B_1 >, & < A_1, B_2 >, & \ldots, & < A_1, B_m > \\ < A_2, B_1 >, & < A_2, B_2 >, & \ldots, & < A_2, B_m > \\ \vdots & \vdots & \ddots & \vdots \\ < A_n, B_1 >, & < A_n, B_2 >, & \ldots, & < A_n, B_m > \end{array}$$

This can be realized by the following straightforward Prolog code:

```
SetExprA ++> SetExprB :-
   Ai .? SetExprA,
   Bj .? SetExprB,
   superpose(Ai,Bj),
   stop_search.
SetExprA ++> SetExprB.
```

The only new predicate we used is `stop_search/0`. In fact, one omitted item in the TPE interface we have observed is a test to control the status of the KB. Therefore, we extend the TPE interface with

• *(TPE.6)* - **proof_found(**$-Int$**)** :

Used to ask the status of the KB. The number of found proof(s) is given.

You can think of `stop_search/0` as built from a `proof_found/1` call followed by an appropriate comparison and by any other (eventually) necessary operations.

In addition to the `++>/2` operator, zSDL also defines the syntax

◇ *(zSDL.5)* - *superposition* :

$+SetExpr_A$ `<+>` $+SetExpr_B$

With the `<+>/2` operator each couple is also reversed (except for the $<X,X>$ ones).

As we commented, the general form of an inference rule in zSDL is thought to be

$$\frac{Id_1, Id_2}{[N_1, N_2, \ldots, N_m]}.$$

The actual application of such a rule is called by

$[Id_1]$ `++>` $[Id_2]$.

The first missing item is a way to get, in a zSDL program, the id-set of the generated formulas. With a typical Prolog attitude, we can generalize this problem.

A superposition goal on id-sets is like evaluating a high-level function on a set. The relation that links the input and the output sets is different from the classical ones, for it is related to some properties of the objects in the sets and not to the sets themselves. This simply implies that the actual module responsible of the evaluation of these relations is not the classical one. And we know that that module must be the TPE. So we are looking for a syntax like

◇ *(zSDL.6)* :

$?Index$ `::=` $+TPE\_Goal$,

where a TPE_Goal can be, as an example, a superposition call. Notice that we defined the new operator `::=/2` in order to switch the evaluation to the right module. The call also suggests a possible model for the computation of the goal. In fact a goal of the TPE is generally requested to produce a new index (say a *dynamic* index) that is updated during the actual evaluation of the goal. Consider the following code.

```
Given ::= TPE_Goal :-
   new_dynamic_index(NewSet),
   call(TPE_Goal),
   (  var(Given),
      Given .. NewSet
   ;  Given := NewSet  ),
   del_dynamic_index(NewSet).
```

It asks the TPE to release a new dynamic index that will be updated during the execution of the given TPE_Goal to hold the result of the evaluation. This result is then properly assigned to the input Given argument and finally the dynamic index is cleared. This asks for the extension of the TPE interface with the two following calls

• *(TPE.7)* - **new_dynamic_index(**$-SetName$**)** :

Ask the TPE to extend the sets of active indexes. *SetName* will be used to refer to this new dynamic id-set. The complementary call is

• *(TPE.8)* - **del_dynamic_index(**$+SetName$**)** :

It is used to remove the index referred by *SetName* from the set of the dynamic indexes known by the TPE.

With the new zSDL operator we can now use the following statement to sketch the application of an inference rule:

$NewIds$ `::=` $[Id_1]$ `++>` $[Id_2]$.

where $NewIds$ will be instantiated to the right instance of $[N_1, N_2, \ldots, N_m]$, even possibly the empty id-set. Notice that the `::=/2` operator works for each TPE goal.

The last extension we will give before going through some examples of an application of zSDL focus on a way to have a local specification of the inference rules we wish to apply in a TPE goal. The zSDL syntax is:

◇ *(zSDL.7)* :

$+TPE\_Goal$ `./` $+Inferences$,

defines a TPE evaluation *modulo* a given set of inference rules.

Suppose for example we wish to superpose clauses 3 and 15 only by binary resolution (`binary_res`). Consider the following code

```
TPE_Goal ./ Inferences :-
   Active .. enabled_inferences,
   disable(Active),
   enable(Inferences),
   call(TPE_Goal),
   disable(Inferences),
   enable(Active).
```

With this new operator we can express the preceding problem as

```
Resolvents ::= [3] ++> [15] ./ binary_res.
```

The code we have given assumes that the `enable/1` and `disable/1` calls in the TPE interface maintain one set, called `enabled_inferences`, collecting the names of the active inference rules.

So in zSDL the more general IP activation call to the TPE is

$$NewIds \ ::= \ Expr_A \ <+> \ Expr_A \ ./ \ Infs.$$

which will give in *NewIds* the id-set of all the formulas derivable by applying the chosen inferences to all the pairs of formulas implicitly referred to by the id-set expressions.

4 A simple zSDL program: the breadth-first strategy

Time has come to give the first example of the use of zSDL to describe a classical strategy: the breadth-first search. We suppose that the TPE is already active and some input formulas are present in the KB. An index called `input` collects the references to those statements.

In the breadth-first search the next level of the tree is filled with all the conclusions given by superposing the last level with all the existing levels. The search stops with complete search or, for example, with a proof. The zSDL program is

```
breadth_first :-
  levels := input,
  last   := input,
  while( ( \+ stop_search,
           \+ last .= [] ),
       ( Next ::= last <+> levels,
         last := Next,
         levels #= last ) ).
```

The two indexes, `levels` and `last`, refer to the entire tree and to its last level, respectively. The `while/2` is the classical cyclic structure you found in each procedural language. Its syntax is

◇ *(zSDL.8)* - **while**(+*Condition*, +*Goal*) .

After the initialization of the values to the input references, the program repeatedly fills the `Next` level of the search tree, superposing the `last` level with all the nodes. Then the `Next` level becomes the `last` and is also added to the references of the entire tree. The `#=` notation resemble the C language style assignments. Similarly zSDL accepts the operators `+=`, `-=` and `*=`. Notice also that the instances of the Prolog variable(s) in the `while/2` statement are released between the cycles.

The preceding algorithm can be improved by thinking of the cases it generates. When we superpose the last level with the entire tree, we must note that all of the nodes in `last` are already in `levels`. Furthermore, if we apply the `<+>` operator to superpose an id-set on itself, we try all of the pairs twice. So, a better program is

```
breadth_first :-
  last   := input,
  others := [],
  while( ( \+ stop_search,
           \+ last .= [] ),
       ( LL ::= last ++> last,
         L0 ::= last <+> others,
         others := last .+ others,
         last := LL .+ L0 ) ).
```

In this definition the `last` index refers again to the last level of the tree while `others` refers to the rest of the tree. At each step `last` is superposed on itself (with the oriented operation `++>`) and then with the upper levels of the tree. You might also note that in this way we can substitute the use of the standard union with the weak one (append) as no repetitions are possible in the references in the indexes.

In addition to the while statement, zSDL defines some other basic control structure:

◇ *(zSDL.9)* - **foreach**(+*Generator*, +*Goal*) :

Goal is executed for all the solutions of the given *Generator*.

◇ *(zSDL.10)* - **repeat**(+*Goal*, +*Condition*) :

Goal is executed at least once and re-executed while *Condition* fails.

◇ *(zSDL.11)* - **iF**(+*Condition*, +*Goal*) :

Goal is executed only if the *Condition* holds. It always succeeds.

This list is given only for completeness: the reader might note that zSDL programs are basically extended Prolog programs and that all the structures definable on the underlying Prolog machine can be used by zSDL programs.

However, we think that one real important aspect of the $< TPE, zSDL >$ Prolog-based architecture comes from its direct executabilty on a Prolog machine. The global proving system loses the property to be batch or interactive: a proof search is directed by the execution of goals, and the granularity of these steps can vary from the single superposition to the entire search.

5 More complex applications

The availability of a language like zSDL adds to the ease of implementing and experimenting with new ideas, for example, non-standard search strategies. To illustrate the value of using of zSDL, and to introduce some additional features of this language, we now focus on three somewhat complex programs. The first defines an *adaptive*, weighting-based, search strategy. The second introduces some atypical *deletion* strategy into the search. The last one shows how to define a strategy (oriented) *tailored* to a given inference rule.

5.1 A weight-based adaptive strategy

By weighting (w) strategies we refer to those algorithms structured to consider the length, or weight, of the formulas. Examples of w-functions are: the number of symbols in a formula, the number of (positive, negative, total) literals in a clause, as well as linear functions built on these or other values. The general behavior of a w-strategy is to filter the retention in the KB of a newly generated formula, according to the given w-function. Formulas that are too heavy are discarded. The underlying intuitive idea is that if a proof can be obtained without the use of heavy formulas, then such formulas can be discarded.

We shall not consider the well-known subproblems that the subsumption operation can lead to, which vary with the w-function adopted. Instead, we consider one of the practical difficulties in the application of these strategies, namely, choosing the appropriate threshold (upper bound on weights) to use for deciding which formulas to discard. The solution we propose follows this simple idea: the threshold can be increased, when the search stops generating formulas, and set to the lightest weight template in the set of the w-deleted formulas. In this sense the search is adaptive: it adapts to the performance of the program.

Let us first show the mechanisms provided by the TPE to support w-strategies. Each formula is stored with a `weight template`. An internal function, namely, `weight(+`*Formula*`,-`*W_Template*`)`, is used by the TPE to calculate it. Such a template consists of a 4-integers tuple ($N-P-T-S$) that counts Negative_Literals, Positive_Literals, Total_Literals and Symbols, where the first three values are "0" if the formula is not a clause. The TPE offers some calls in order to define weighting-based strategies:

- *(TPE.9)* - **max_weights**(?*W_Template*) :

  The call can be used both to access the current reference w-template (if *W_Template* is an uninstantiated variable at the call) or to set a new value for it. The new given *W_Template* will be used by the w-filter operation to decide which new formulas to accept or discard. All the values for the new formulas must be less or equal to the threshold ones fixed by the given *W_Template*. The value of a variable will be considered greater than each integer.

- *(TPE.10)* - **formula_weight**(+*Id*, −*W_Template*) :

  Accesses the given formula(s) to get their weight template(s).

The basic behavior of the strategy we are going to write is straightforward. At each time we choose the lightest not yet used formula in the KB to be superposed with all the already used ones. Than we move the given formula to the set of the used ones (say "done") while the new generated formulas are added to the first set (say "to_do"). We can express this with the following zSDL program:

```
to_do := [],
done  := [],
Input .. input,
add_ordered(Input,to_do),
while( ( \+ stop_search,
         \+ to_do .= [] ),
     ( Lightest .~ to_do,
       add_ordered([Lightest],done),
       New ::= [Lightest] <+> done,
       add_ordered(New,to_do) ) ).
```

As one sees, we solved the problem of getting the lightest formula in a set by extracting the first element from an ordered set. The expected side effect of an `add_ordered(`*Set*`,`*SetName*`)` call is to build an ordered union of *Set* and *SetName* (into *SetName*) according to the weight of the corresponding formulas. We can obtain this with:

```
add_ordered([],_SetName).
add_ordered(Set,SetName) :-
   Xet .. SetName,
         %% gets a list of Count-Id pairs
   get_counts(Set,SetCs),
   get_counts(Xet,XetCs),
   append(SetCs,XetCs,YetCs),
         %% sorts by counts
   keysort(YetCs,ZetCs),
         %% removes the counts
   pop_counts(ZetCs,Zet),
   SetName := Zet.
```

where the `get_counts/2` call accesses the weights-template of the formulas to get the symbol counts (obviously, one can choose different approaches).

To extend our strategy to be self-adaptive we have to solve certain problems:

- how to get information on the deleted formulas;
- how to choose some initial value for the reference w-template.

The first problem rests entirely on the TPE behavior, as the "over-weight" deletions are embedded into its operations. Our system maintains a set of structures, indexed by weights-template, to have the references to the deleted formulas. The call

- *(TPE.11)* - `queue(wdel(?W_Template),?Queue)` :

  Queue holds the ids of all the deleted formulas sharing the same it W_Template.

We first give the extended program that realizes the self-adaptive search, and then we discuss its main steps.

```
self_adaptive :-
   input_weighting,
   to_do := [],
   done  := [],
   Input .. input,
   add_ordered(Input,to_do),
   while( ( \+ stop_search,
            (  \+ to_do .= []
            ;  q_exists(wdel(_))  ) ),
     once (  to_do .= [],
             lightest_deleted(Count),
             closest_wtemplate(Count,NewWT),
             max_weights(NewWT),
             deleted := [],
             add_deleted(Count,deleted),
             Unhide .. deleted,
             Restored ::= undelete(Unhide),
             add_ordered(Restored,to_do)
             Lightest .~ to_do,
             add_ordered([Lightest],done),
             New ::= [Lightest] <+> done,
             add_ordered(New,to_do) ) ).
```

The first difference concerns the while condition: it now considers the possible presence of formulas deleted by weight, so the search is complete only if no deleted formulas remain. The `lightest_deleted/1` call accesses the deletion queue, searching for the lightest-weight formula. Its definition can be:

```
lightest_deleted(Count) :-
   setof( SymCount,
          queue(wdel(N-P-L-SymCount),Q),
          Deleted ),
   sort(Deleted,[Count|_Others]).
```

The `closest_wtemplate/2` call is responsible for deciding the value for the new reference weights-template, or, in other words, for the "size of the adaptation-step". The following definition builds the new template in order to accept all the formulas with the given deleted smallest *symbol* count.

```
closest_wtemplate(Count,Template) :-
   setof( N-P-L-Count,
          queue(wdel(N-P-L-Count),Q),
          Deleted ),
   max_weights(CurrentWT),
   max4([CurrentWT|Deleted],Template).
```

where the call to `max4/2` builds the `Template` given by the maximal values for each count.

The `add_deleted/2` call is conceptually similar to the `add_ordered/2` call, but works with the deletion queue. Its definition can be:

```
add_deleted(Count,SetName) :-
   (  queue(wdel(N-P-L-Count),Queue),
      SetName += Queue,
      q_del(wdel(N-P-L-Count)),
      fail
   ;  true  ).
```

It collects into `SetName` all the references to the deleted formulas with the given symbol `Count` and deletes the corresponding queue (`q_del/1`).

So, in the while loop of our program, the `to_do` id-set is extended either by newly inferred formulas or by reactivating the lightest deleted ones (if any).

A last point addresses the choice of the initial values for the reference w-template. A strategy that has given us interesting results fixes the values by looking at the counts of the input formulas and choosing the lowest values among them. Its definition is:

```
input_weighting :-
   Input .. input,
   formula_weight(Input,WTs),
   max4(WTs,Template),
   max_weights(Template).
```

5.2 A pruning strategy

This second example of the applications of the zSDL language is given to show how it can be used to define some self-analytical activity for the proving process. In other words we can use it to reason about the current state of the search during the execution.

A well-known problem each ATP program must face is the possible explosion of the search space, which can occur for various reasons. Here we do not study this topic, nor do we suggest that our program has a deep impact on the solution of the general problem. Our goal is only to show how an SDL language can be useful in different research areas of ATP.

We observe that our pruning strategy is based on the addresses of the *undeterminism in the order of application of the inference steps*. On the other hand, the use of reduction rules comes from the wish to have a KB capture the same logical consequences with a smaller possible representation "size". Consider now a generic search process and suppose a reduction step occurs. With "reduction" we will refer to the results of an operation able to change the structure of a formula, maintaining its logical value. Generally speaking a reduction step will reformulate a formula by "reducing" its complexity and/or size. This transformation will in general involve other formulas used as a base for the logical reformulation. As an example, consider the following steps on two generic clauses

$$\frac{\dfrac{[1]\,\neg A \mid B, [2]\,\neg A \mid \neg B \mid C}{[3]\,\neg A \mid C}\ \textit{binary resolution}}{\textit{delete 2}}\ \textit{subsumption}$$

We can view this step as the application of a reduction rule that uses [1] to transform [2] into [3]. We note that the satisfiability of the overall KB is preserved, i.e. the operation maintains the logical truth of the set of formulas.

Suppose next that such a reduction has occurred during a search, say a formula $\mathcal{F}$ has been reduced to $\mathcal{F}'$. There now exists a potential set of formulas whose generation depends on the order in which the search process has been executed: this set consists of all of the descendants of $\mathcal{F}$ that have not contributed to the generation of $\mathcal{F}'$, or, more precisely, the set

$$\textit{by\_inference(descendants}(\mathcal{F})) \textit{ - ancestors}(\mathcal{F}').$$

(We note that we must leave all the descendants of $\mathcal{F}$ given by reduction as those are formulas originally not in the set generated by $\mathcal{F}$).

Pruning this set (if not empty) could perhaps make the proof longer, as the proof could be reachable rapidly by using one of the formulas we deleted, but it will not preclude the possibility of finding the proof if there is one.

The effectiveness of this pruning strategy depends mainly on the effectiveness and the applicability of reduction steps in a proof, and so it relies directly on the structure of the search space (given by the formulas asserting the theorem).

Let us now see how we can implement this operation by using zSDL and the mechanisms of the TPE. First of all we formalize the calls the TPE defines (and zSDL inherits) to access various relations on the content of the KB. We already announced some of them in section 2.

- *(TPE.12)* - **parents**$(+Id, -Parents)$
- *(TPE.13)* - **ancestors**$(+Id, -Ancestors)$
- *(TPE.14)* - **children**$(+Id, -Children)$
- *(TPE.15)* - **descendants**$(+Id, -Descendants)$:

  Being *Id* the reference to a formula, these calls will respectively return the id-set of its parents, ancestors, children, and descendants, with respect to the current KB. We note that the given id-set may contain references to currently inactive formulas (deleted for some reason). All these relations will consider both inference as well as reduction steps.

- *(TPE.16)* - **by_reduction**$(+IdSet, -ByRed)$:

  Given an *IdSet* this call selects which referred formulas have been produced by application of a reduction rule, building the id-set *ByRed* with their ids.

- *(TPE.17)* - **replace**$(?NewId, ?Id)$:

  The call succeeds if *NewId* refers to a formula that replaces an old one (referred by *Id*) following a reduction step. Otherwise the call fails.

The proposed pruning strategy acts like a filter on the result of a superposition call: at each step it checks if the new formulas are given by reduction, in which case it tries to apply the deletion. So, we are going to extend the superposition control level of zSDL with a meta-call realizing the pruning.

```
pruning_derive(SetA,Mode,SetB) :-
   XA .? SetA,
   XB .? SetB,
   once (
      Given ::= derive([XA],Mode,[XB]),
      by_reduction(Given,ByRed),
      foreach( NId .? ByRed, (
         replace(NId,Id),
         ancestors(NId,NIdAnc),
         descendants(Id,IdDes),
         by_reduction(IdDes,IdDesByRed),
         IdDesByInf .. IdDes .- IdDesByRed,
         DelSet .. IdDesByInf .- NIdAnc,
         delete(DelSet) ) ) ),
   stop_search.
pruning_derive(SetA,Mode,SetB).

derive(SetA,(<+>),SetB) :- SetA <+> SetB.
derive(SetA,(++>),SetB) :- SetA ++> SetB.
```

The schema is quite simple. Each by-reduction child (`NId`) of a superposition call is related to the formula it replaces (`Id`). Then the set of the by-inference descendant of `Id` is reduced by the set of the `NId` ancestors. Notice how the *by_inference(descendant*$(\mathcal{F})$) set is evaluable as *descendants*$(\mathcal{F})$ *- by_reductions(descendants*$(\mathcal{F})$*)*.

5.3 A hyperresolution-oriented search strategy

Our last example uses zSDL to define a strategy specifically oriented to work with a given inference rule, namely, the inference rule hyperresolution.

The efficiency of an ATP system comes from the efficiency of all of the different components of the program, from the basic unification and match algorithms to the KB management, and so on. With some "tough" inference rule, it also heavily relies on the ability of the search strategy to control its application ensuring a complete search without repeating steps. Hyperresolution is one such inference rule.

Hyperresolution considers a basic clause (called nucleus) that has one or more negative literals. An inference step occurs when a set of positive unit clauses (called satellites) is found that simultaneously unify with all of the negative literals of the nucleus. It is simple to see how hyperresolution will not generate new nuclei (for the rule cannot produce a clause containing negative literals) while it can generate new satellites. So, the set of potential satellites change dynamically during the search, and a good strategy must ensure a complete covering partition (with multiple occurrences) of this set without repeating trials.

We first explain how we implemented the hyperresolution inference rule in our system (we call it hy_p). As usual the rule has two arguments: the first must be a satellite and the second a nucleus. If a unification is found between the given satellite and one of the negative literals in the nucleus, then the set of the current active satellites is partitioned and superposed on the remaining negative literals.

This behavior suggests the development of a search strategy driven by the generation of new satellites. In fact, we can visit the search space by levels, generate all the possible hyperresolvents, choose from them the new satellites, and use those to drive the search in the next level. As those satellites are new, the partitions we will try are new too, and no repetition in the trials occur. The basic shape of the strategy can be:

```
hyper_strategy :-
    Input .. input,
    get_satellites(Input,Sats),
    get_nuclei(Input,Nucs),
    last_sats := Sats,
    nucs := Nucs,
    while( ( \+ stop_search,
             \+ last_sats .= [] ), (
       New ::= last_sats ++> nucs ./ hy_p,
       get_satellites(New,NewSats),
       last_sats := NewSats ) ).
```

The `get_satellites/2` and `get_nuclei/2` calls are used to choose from an id-set the subset of formula-references corresponding, respectively, to valid satellites and nuclei. Notice how these calls can be defined by using the `formula_weight/2` call and testing the negative and positive literal counts accordingly.

As a matter of fact, the algorithm we have given follows closely the general schema of a breadth-first search. So, it can be simply extended to consider the application of more inference rules, intermixing the searches with the control, the `enable/1`, and the `disable/1` operations permitted.

6 Conclusions

This work introduces the concepts of Theorem Proving Engine and Strategy Description Language as architectural and applicative tools in the design and use of an automated theorem-proving program.

The definitions we give reflect a running Prolog system, named zEN2, and,, because of this fact, they inherit a Prolog style structure.

Particular emphasis is given to the use of an SDL as a research tool as well as a way to reinterpret the use of a theorem prover as a batch or as an interactive program. In fact, the availability of an interpreter for such a language offers the possibility of having a system able to cover both of these usages, giving to the user some way of choosing the granularity of the steps the prover must take.

Three examples are given to show the possible application of an SDL. Their purpose is to show its usefulness for expressing and testing new ideas. Some interesting capabilities of zSDL are applied to highlight how it allows the treatment of self-analysis on the state of the search space. Examples of these are the definition of the self-adaptive search and the pruning strategy.

Acknowledgments

The author is very grateful to Larry Wos, Bill McCune and Gianni Degli Antoni for their comments. This work was partially supported by the CEE ESPRIT2 KWICK Project and partially by a grant of the Italian Research Council. Most part of the work was done while the author was visiting the Mathematics and Computer Science Division of the Argonne National Laboratory.

References

[Henschen *et al.* 1974] L.Henschen, R.Overbeek and L.Wos. A Theorem Proving Language for Experimentation. *Communications of the ACM*, Vol. 17 No. 6 (1974)