SYNAPSE, A System for Logic Program Synthesis from Incomplete Specifications

Pierre Flener[1] and Yves Deville
Unité d'Informatique, Université Catholique de Louvain
Place Ste Barbe 2, B – 1348 Louvain-la-Neuve, Belgium
{pf,yde}@info.ucl.ac.be, ✆ + 32 / 10 / 47-{2415,2067}, FAX + 32 / 10 / 45.03.45

The SYNAPSE system is part of FOLON, an integrated logic programming environment (described elsewhere in this volume). It aims at automated logic program synthesis from incomplete specifications. After introducing the idea of incomplete specifications, the SYNAPSE approach to synthesis is described, and the SYNAPSE system is illustrated in terms of a sample execution.[2]

1 Incomplete Specifications

A logic program development methodology has been proposed by the second author [3]. It aims at programming-in-the-small, and is (mainly) meant for "algorithmic" problems. It starts from a complete, yet informal, specification of the target problem. A crucial step is the design of the logic of the program, based solely on the declarative semantics of logic, and with exclusive concern about correctness issues. Procedural semantics and efficiency are taken care of in a later optimization and implementation step.

The methodology gives rise to many computer-assistance, if not automation, opportunities. The FOLON environment [9] aims at this, and is described elsewhere in this volume. But, as specifications are non-formal, the crucial step cannot be automated. This research thus investigates an alternative approach to logic program development, namely (fully) automated logic program synthesis. This requires formal specifications, and our choice went towards exploring synthesis from incomplete specifications. The resulting system, called SYNAPSE, is being integrated into the FOLON environment.

Let $\mathcal{R}$ be the relation one has in mind when elaborating a specification of a procedure for predicate *r*. We call $\mathcal{R}$ the *intended relation*, in contrast to the relation actually specified, called the *specified relation*. This distinction is important in general, but crucial with incomplete specifications, where one deliberately admits a gap between the two. We assume $\mathcal{R}$ is known, even if we don't have a formal definition of it. In our approach [5], incomplete specifications are expressed with examples and properties. More precisely, a *specification by examples and properties* of a relation *r* consists of:

- a set of examples of *r* (ground atoms); and
- a set of properties of *r* (non-recursive Horn clauses).

The specified relation, that is the set of logical consequences of the given examples and properties, is assumed to be a subset of the intended relation $\mathcal{R}$.

Examples have been around for quite a while as an incomplete specification formalism. Their attractiveness lies in their naturalness and conciseness, but their weaknesses are ambi-

1. Supported by the Government of Luxembourg, Ministry of Cultural Affairs and Scientific Research, Grant BFR 92/017.
2. SYNAPSE stands for "SYNthesis of logic Algorithms from PropertieS and Examples".

guity and a limited expressive power. The introduction of properties aims at overcoming these drawbacks, while preserving these strengths. Let’s illustrate this on a sample problem.

Example 1-1: Let the *firstPlateau(L,P,S)* relation hold iff list P is the first maximal sequence of identical elements (plateau) of non-empty list L, and list S is the corresponding suffix of L. A sample specification by examples and properties is:

```
firstPlateau([a],[a],[])                                 (E1)
firstPlateau([b,b],[b,b],[])                             (E2)
firstPlateau([c,d],[c],[d])                              (E3)
firstPlateau([e,f,g],[e],[f,g])                          (E4)
firstPlateau([h,i,i],[h],[i,i])                          (E5)
firstPlateau([j,j,k],[j,j],[k])                          (E6)
firstPlateau([m,m,m],[m,m,m],[])                         (E7)
firstPlateau([X],[X],[]).                                (P1)
firstPlateau([X,Y],[X,Y],[])     ← X=Y                   (P2)
firstPlateau([X,Y|T],[X],[Y|T]) ← X≠Y                    (P3)
```

A sample (normalized) logic program is:

```
firstPlateau(L,P,S) ← L=[HL],
                      P=L,S=[]
firstPlateau(L,P,S) ← L=[HL1,HL2|TL],
                      HL1≠HL2,
                      P=[HL1],S=[HL2|TL]
firstPlateau(L,P,S) ← L=[HL1,HL2|TL],
                      HL1=HL2,
                      firstPlateau([HL2|TL],TP,TS),
                      P=[HL1|TP],S=TS
```

Properties P_1 to P_3 generalize the examples of $\{E_1\}$, $\{E_2\}$, and $\{E_3, E_4, E_5\}$, respectively. It is easily apparent how this specification improves upon its examples-only counterpart: properties allow the specifier to make explicit what s/he perfectly knows, but can’t express by examples alone. Especially P_3 embodies such additional knowledge that has otherwise to be guessed by the synthesizer, which is dangerous (risk of wrong guesses) and time-consuming (enumeration of all possible guesses). The very presence of properties is thus expected to increase reliability and efficiency of synthesis compared to an examples-only approach.

The use of the SYNAPSE system is interesting because it generates, from specifications that are extremely easy to write, a correct logic program in a (nearly) automatic way.

2 The SYNAPSE Approach to Synthesis

Programs can be classified according to their design strategies (such as divide-and-conquer, generate-and-test, global search,…). It is therefore interesting to guide a design process by a program schema (template program with fixed control flow) that captures the essence of such a strategy. In this research, we do so, and focus on the divide-and-conquer strategy.

Loosely speaking, a *divide-and-conquer program* for a binary predicate r over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is found by directly solving the problem. Otherwise, if X is non-minimal, decompose X into a vector $\boldsymbol{HX}$ of heads of X and a vector $\boldsymbol{TX}$ of tails of X, the latter being of the same type as X, as well as smaller than X according to some well-founded relation. The tails $\boldsymbol{TX}$ recursively yield tails $\boldsymbol{TY}$ of Y. The heads $\boldsymbol{HX}$ are processed into a vector $\boldsymbol{HY}$ of heads of Y. Finally, Y is composed from its heads ***HY*** and tails ***TY***. Recursion is sometimes useless, namely when *Y* can already be directly computed from ***HX*** and ***TX***. Moreover, one has to discriminate between the various non-minimal cases according to the values of ***HX***, ***TX*** and *Y*.

Logic program schemata can be expressed as second-order logic programs. For instance, many logic programs designed by a divide-and-conquer strategy fit the following schema:

```
R(X,Y) ←  Minimal(X),
          Solve(X,Y)
R(X,Y) ←  NonMinimal(X),
          Decompose(X,HX,TX),
          Discriminate1(HX,TX,Y),
          SolveNonMin(HX,TX,Y)
R(X,Y) ←  NonMinimal(X),
          Decompose(X,HX,TX),
          Discriminate2(HX,TX,Y),
          R(TX,TY),
          Process(HX,HY),
          Compose(HY,TY,Y)
```

where **`R(TX,TY)`** denotes a conjunction of atoms `R(TX`$_i$`,TY`$_i$`)`. There can be any number of clauses of the second and third kind.

The SYNAPSE Synthesis Steps

A unique, generalized version (for relations of any arity) of this schema underlies our synthesis mechanism, which can then be expressed as the following fixed sequence of steps:

- Step 1 – Syntactic creation of a first approximation (Instantiation of *R*);
- Step 2 – Synthesis of *Minimal* and *NonMinimal*;
- Step 3 – Synthesis of *Decompose*;
- Step 4 – Synthesis of the conjunction of recursive atoms;
- Step 5 – Synthesis of *Solve* and *SolveNonMin*;
- Step 6 – Synthesis of *Process* and *Compose*;
- Step 7 – Synthesis of the *Discriminate*$_k$.

Another requirement for the synthesis mechanism is a non-incremental presentation of the specification, that is the specifier has to provide all examples and properties prior to synthesis. We also strive for a synthesis mechanism that performs both inductive and deductive reasoning. This ensures that both examples and properties take a constructive role during synthesis. In other words, examples are not to be used as test-data for a purely deductive synthesis from the properties, and properties are not to be used as integrity constraints for a purely inductive synthesis from examples.

3 A Sample Synthesis with the SYNAPSE System

A synthesis mechanism following the above requirements has been identified [6] [7], and is currently being implemented as the SYNAPSE system. A first prototype exists. We illustrate its performance on the *firstPlateau* problem. Unless otherwise noted, the user has nothing to do once s/he has introduced the specification. Moreover, s/he doesn't even see the intermediate versions.

Step 1 – Syntactic creation of a first approximation (Instantiation of *R*)

Step 1 instantiates *R* with the predicate used in the specification, and generates the fact "*r(X,Y)* ←" as a first approximation that is satisfied by all examples. In our case, we obtain:

```
    firstPlateau(L,P,S) ←                                  {E1-E7}
```

The set annotation explains which examples are covered by a clause.

Step 2 – Synthesis of *Minimal* and *NonMinimal*

Step 2 selects an induction parameter, and selects a minimal and a non-minimal form for this parameter from a type database. The examples are thus partitioned into two classes, according to the form they actually satisfy. Prior to synthesis, the user may hint at a preference for an induction parameter. In our case, supposing *L* is selected as induction parameter, the initial clause is replaced by the following two clauses:

```
    firstPlateau(L,P,S) ← L=[_]                              {E1}
    firstPlateau(L,P,S) ← L=[_,_|_]                       {E2-E7}
```

Indeed, in the examples, parameter *L* is either a singleton list, or a list of at least 2 elements.

Step 3 – Synthesis of *Decompose*

Step 3 also uses the type database to decompose, in the non-minimal clause, the induction parameter into a vector of heads and a vector of tails, the latter being each smaller than the induction parameter according to some well-founded relation. Sample decomposition strategies for lists are head-tail decomposition, splitting in halves, partitioning, and so on. Prior to synthesis, the user may hint at a preference for a decomposition strategy. In our case, assuming a head-tail decomposition is used, the non-minimal clause becomes:

```
    firstPlateau(L,P,S) ← L=[_,_|_],
                          L=[HL|TL]                       {E2-E7}
```

Step 4 – Synthesis of the conjunction of recursive atoms

Step 4 introduces, in the non-minimal clause, a recursive atom for the tail *TL* of the induction parameter. This yields tails *TP* and *TS* of the other parameters. A look-ahead check is performed, for each example covered by the non-minimal clause, to see whether the values of *TP* and *TS*, deduced by using the specification as an oracle, are actually "used" in the construction of *P* and *S*. For instance, in example E_2, *TL* is [*b*]. By property P_1, the computed *TP* and *TS* are [*b*] and [], respectively, which are "used" in the construction of *P* and *S*, respectively. But, in example E_3, *TL* is [*d*], and the computed *TP* is [*d*], which is not "used" in the construction of *P*. Recursion is thus not always useful, so the non-minimal clause is split into a non-recursive one and a recursive one:

```
    firstPlateau(L,P,S) ← L=[_,_|_],
                          L=[HL|TL]                       {E3-E5}
    firstPlateau(L,P,S) ← L=[_,_|_],
                          L=[HL|TL],
                          firstPlateau(TL,TP,TS)      {E2,E6,E7}
```

Step 5 – Synthesis of *Solve* and *SolveNonMin*

Step 5 completes the minimal case and the non-recursive non-minimal clause by synthesizing a formula that constructs the other parameters from the induction parameter. This is a variant of Step 6. For our sample problem, this is straightforward, and the 2 clauses now are:

```
firstPlateau(L,P,S) ← L=[_],
                      P=L,S=[]                          {E1}
firstPlateau(L,P,S) ← L=[_,_|_],
                      L=[HL|TL],
                      P=[HL],S=TL,TL=[_|_]            {E3-E5}
```

Step 6 – Synthesis of *Process* and *Compose*

Step 6 synthesizes, for the recursive clause, a formula that (i) processes the head *HL* into the heads *HP* and *HS*, and (ii) composes *P* and *S* from *HP*, *TP* and *HS*, *TS*, respectively. This can be done simultaneously by looking for a formula that computes *P* from *HL*, *TP*, and *S* from *HL*, *TS*. The used method, called the *MSG Method*, computes the most-specific generalization (msg) of the <*HL*,*TP*,*P*,*TS*,*S*> tuples extracted from the recursive clause. For our sample problem, the involved values and msg are given in the following table:

	HL	TP	P	TS	S
E_2	b	[b]	[b,b]	[]	[]
E_6	j	[j]	[j,j]	[k]	[k]
E_7	m	[m,m]	[m,m,m]	[]	[]
msg	[A]	[A|T]	[A,A|T]	U	U

The msg is rewritten as a conjunction of atoms, and inserted into the recursive clause:

```
firstPlateau(L,P,S) ← L=[_,_|_],
                      L=[HL|TL],
                      firstPlateau(TL,TP,TS),
                      P=[HL|TP],S=TS,TP=[HL|_]  {E2,E6,E7}
```

In case the MSG Method is not powerful enough, say when processing and composing needs a full-fledged recursive program by itself, then the system calls itself recursively with an inferred specification by examples and properties of the sub-problem.

Step 7 – Synthesis of the *Discriminate$_k$*

Step 7 completes the non-minimal clauses by synthesizing formulas explaining when each of these clauses is applicable. These formulas are discriminants, but need not be mutually exclusive, as logic programs may be non-deterministic. The used method, called the *Proofs-as-Programs Method*, adds literals to the current logic program such that the properties are logical consequences thereof. This is done by trying to prove that each property is a logical consequence of the current program: if yes, nothing is done; otherwise, appropriate literals are extracted from an explanation of the failure. For our sample problem, this goes as follows:

- property P_1 is (unconditionally) provable from the minimal clause;
- property P_2 is provable from the recursive, non-minimal clause under the condition:

  `L=[A,B|T],A=B`

- property P_3 is provable from the non-recursive, non-minimal clause, provided:

  `L=[A,B|T],A≠B`

These conditions are rewritten, and inserted as discriminants into the appropriate clauses:

```
firstPlateau(L,P,S) ← L=[_,_|_],
                      L=[HL|TL],
                      TL=[H|T],HL≠H,
                      P=[HL],S=TL,TL=[_|_]        {E3-E5}
firstPlateau(L,P,S) ← L=[_,_|_L],
                      L=[HL|TL],
                      TL=[H|T],HL=H,
                      firstPlateau(TL,TP,TS),
                      P=[HL|TP],S=TS,TP=[HL|_]  {E2,E6,E7}
```

The synthesis is now terminated. The obtained program is equivalent to the one given above. Alternative, but still correct, programs may be obtained by re-considering the decisions of Steps 2 and 3.

4 Evaluation

The SYNAPSE system synthesizes logic programs from examples and properties, in a non-incremental and schema-guided way. It is part of the FOLON environment. It is modular in that the methods used to synthesize instantiations of predicate variables from the schema are highly interchangeable, and that new methods can easily be integrated. The database of type-specific instantiations of *Minimal*, *NonMinimal*, and *Decompose* can be extended at will. This, together with the concept of properties and the possibility of nested syntheses, is our solution to the predicate invention problem. Our experience with the system has shown the viability of specifications by examples and properties. Synthesis is quite efficient and reliable. The synthesized programs are independent of example or property ordering. A methodology of choosing "good" examples and properties is being formulated.

This work is part of the currently emerging field of *inductive logic programming*, which aims at upgrading the techniques of the classical empirical machine learning paradigm into a logic programming framework [11] [12]. The main differences are that we only consider recursive programs, whose intended relations are furthermore known. This motivates the non-incremental, schema-guided approach. The idea of using properties (not to be confused with background knowledge) is not unique [2] [4], but we are not aware of any approaches that use properties constructively. The use of schemata is well-established in program design [1] [8] [10] [13] [14] [15] [16] [17].

The divide-and-conquer schema is hard-wired into SYNAPSE: as the developed methods are very general, the support and selection of user-provided schemata is envisaged.

Acknowledgments

We gratefully acknowledge many insightful discussions with Baudouin Le Charlier and the other members of the FOLON Research Project (University of Namur, Belgium).

References

[1] J. Burnay and Y. Deville. Generalization and program schemata: A step towards computer-aided construction of logic programs. In *Proceedings of NACLP'*89, pages 409–425. MIT Press.

[2] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.

[3] Y. Deville. *Logic Programming: Systematic Program Development*. International Series in Logic Programming, Addison Wesley, 1990.

[4] W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and MH. Rogers (editors), *Proceedings of META'*88, pages 501–521. MIT Press.

[5] P. Flener. Towards programming by examples and properties. TR CS-1991-09, Duke University, Durham (NC, USA), 1991.

[6] P. Flener and Y. Deville. Towards stepwise, schema-guided synthesis of logic programs. In T. Clement and KK. Lau (editors), *Proceedings of LOPSTR'*91, pages 46–64. Springer Verlag, 1992.

[7] P. Flener and Y. Deville. Synthesis of composition and discrimination operators for divide-and-conquer logic programs. In JM. Jacquet (editor), *Proceedings of the ICLP'*91 *Workshop on Logic Program Construction*. John Wiley, 1992. (In print.)

[8] TS. Gegg-Harrison. *Basic Prolog schemata*. TR CS-1989-20, Duke University, Durham (NC, USA), 1989.

[9] J. Henrard and B. Le Charlier. FOLON: An environment for declarative construction of logic programs. In M. Bruynooghe and Wirsing (editors), *Proceedings of PLILP'*92, pages 217–231. LNCS 631, Springer Verlag.

[10] A. Lakhotia. Incorporating "programming techniques" into Prolog programs. In *Proceedings of NACLP'*89, pages 426–440. MIT Press.

[11] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8(4):295–317, 1991.

[12] EY. Shapiro. *Algorithmic Program Debugging*. PhD-Thesis, Yale University, 1982. Published under the same title by MIT Press, 1983.

[13] DR. Smith. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, 27:43–96, 1985.

[14] DR. Smith. The structure and design of global search algorithms. TR KES.U.87.12, Kestrel Institute, Palo Alto (CA, USA), 1988.

[15] LS. Sterling and M. Kirschenbaum. Applying techniques to skeletons. In JM. Jacquet (editor), *Proceedings of the ICLP'*91 *Workshop on Logic Program Construction*. John Wiley, 1992. (In print.)

[16] P. Summers. A methodology for LISP program construction from examples. *Journal of the ACM*, 24(1):161–175, January 1977.

[17] NL. Tinkham. *Induction of Schemata for Program Synthesis*. PhD-Thesis, Duke University, 1990.