

Applications of Logic Programming in Software Engineering

P.Ciancarini
University of Bologna
cianca@cs.unibo.it

G.Levi
University of Pisa
levi@di.unipi.it

Abstract

We show that logic programming offers useful methods and techniques to software engineers. Several research and industrial projects have either successfully applied logic programming languages during the software development lifecycle, or have developed useful software engineering tools exploiting some feature of logic programming. We overview the main software specification and design techniques based on Prolog or on more advanced logic languages. We also survey a number of software development tools based on logic programming. These tools range from constraint-based structure editors, symbolic debuggers, and (de)compilers, to configuration tools, project databases, and planners for knowledge-based programming-in-the-large, to rule-based notations and process-centered environments for coordinating and enacting the software development process.

Index Terms: Software Engineering, Formal Methods, Specification, Design, Testing, Logic Programming, Software Prototyping, Maintenance, Configuration Management, Software Process Modeling.

Contents

1	Introduction	3
2	Main features of Prolog from the perspective of a software engineer	4
2.1	Prolog for the software specifier	4
2.1.1	Prolog as specification notation	4
2.1.2	Prolog as support for rapid prototyping	6
2.1.3	Prolog as specification animation system	7
2.1.4	Prolog as notational integrator	8
2.2	Prolog for the software designer	9
3	Beyond Prolog: Constraints, Concurrency, Objects	11
3.1	Constraint Logic Programming	12
3.2	Concurrent Logic Programming	13
3.3	Object-oriented Logic Programming	14
3.4	Discussion	16
4	Applications of Logic Programming in Software Engineering	16
4.1	Programming in-the-small	17
4.1.1	Compilers	17
4.1.2	Structure editors	18
4.1.3	Symbolic Debugging and Analysis Tools	19
4.1.4	Testing	20
4.2	Programming-in-the-large	21
4.2.1	Configuration Management and Version Control	21
4.2.2	Maintenance and Reuse	23
4.3	Programming-in-the-many	24
4.3.1	Project Management	24
4.3.2	Software Process Modeling	24
5	Discussion	27
6	Conclusions	29

1 Introduction

The end of the Japanese Fifth Generation project [105, 168] has marked a crucial point in the history of logic programming. The evaluation of the results of such a project has generated a considerable debate centered around the effective applicability of the logic programming technology in industrial contexts, see for instance [195]. It has been argued that logic programming will survive only insofar as useful applications can be built and proved effective, with the implication that such applications have not yet been built. According to these opinions, no “killer” application currently demonstrates the usefulness of logic programming in industrial practice.

A relevant feature of the debate is the influence of logic programming on software engineering research and practice. Most software engineers would say that such an influence is small. However, in an era of increasing software costs, software engineers need to investigate new methods and techniques to reduce the costs and the complexities of the software lifecycle, and to face and solve problems encountered in building new software systems or adapting old ones (legacy systems). In this paper we show which features of logic programming are most useful to software engineers, and which software engineering applications have been successfully developed, based on a logic programming approach.

A caveat is necessary: we do not intend to suggest that logic programming is a “silver bullet” [28], *i.e.*, a panacea good for every software engineering problem, but only to show how such a technology has been extensively used in all phases of the software lifecycle, and has produced useful tools for the practitioner. Thus, we do not make any comparison with other paradigms, such as functional or object-oriented programming; we do not intend to defend any thesis of the kind “language X is better than language Y for software engineering task Z”; we simply hope to offer to our readers a reference to possible solutions to practical software engineering problems. Moreover, we restrict our discussion to software engineering applications in the sense that we will exclude any reference to the two most successful application fields of logic programming: expert systems and deductive databases, except when the application field is software development and maintenance.

Logic programming is quite popular in universities, especially in Europe. On the other hand, industrial projects that exploit logic programming are not uncommon today; there is even a conference devoted to “real” applications of logic programming [141]. Among logic languages, the most widely known and used is of course Prolog, thanks to its flexibility and availability of efficient implementations. Probably the reason of such a diffusion is the fact that Prolog can be used in different applicative contexts as an economical substitute for more specific tools and languages. In several situations it is the second best choice, after the special languages developed on purpose for a specific domain. For instance, modern Prolog systems can be used as data modeling languages, as system programming environments [65], as executable specification languages, as constraint programming languages, as toolkits to build graphical user interfaces, as tools for reverse engineering, and even as learning environments for teaching beginners programming [164].

The variety of these applications shows that Prolog is very flexible and general-purpose, but how could it be effectively used by a software engineer?

Any software development process is conventionally split into several phases, of which the most important are specification, design, coding, testing, and maintenance. We will see that Prolog has been successfully used in all the phases of the development process. This is mainly due to the progress of logic programming technology, which is currently able to support the implementation of efficient Prolog executors and environments, and the evolution of software development environments technology, which is gradually moving from a tool based approach towards a knowledge based approach [156]. In fact, we have found several papers by software engineers describing experiments in which Prolog-based tools in programming-in-the-small and programming-in-the-large were successfully developed and tested. For these activities, the most common software tools based on logic programming are compilers and configuration management systems. There is also a third class of tools successfully based on logic programming, namely for programming-in-the-many, which coordinate several cooperating software engineers: rule-based process-centered environments [3, 155].

Logic programming should not be identified with Prolog; there is something else. In fact, some new trends in logic programming research, *i.e.*, constraint logic programming, concurrent logic programming,

and object-oriented logic programming, are broadening the spectrum of applications of logic programming from the perspective of software engineers. As examples, we cite the use of constraint languages in the development of specialized problem solvers [55, 192] and in visual programming [89], the use of concurrent logic languages as specification languages for system modeling [58], and the use of logic languages extended with object-oriented features in the design of innovative programming environments [131].

The paper is structured as follows: in Section 2 we review the main features of logic programming languages, in particular Prolog, from the perspective of software specifiers and designers. In Section 3 we summarize the main features of constraint logic programming, concurrent logic programming, and object oriented logic programming, and discuss their implications on software engineering practice. In Section 4 we review a number of existing software engineering applications, classifying them into three categories: programming-in-the-small, programming-in-the-large, and programming-in-the-many. In Section 5 we discuss when and why a software engineer should consider using logic programming technology. Finally, Section 6 sketches some issues worth of considering for future research.

2 Main features of Prolog from the perspective of a software engineer

The most widespread logic language is Prolog, so we start our discussion from this language. Which are the relevant features of Prolog from the perspective of a software engineer?

Logic languages have been proved adequate for applications like deductive databases, expert systems, rule-based AI applications, and more generally as general-purpose tools for declarative and non deterministic programming. Being a logic language, Prolog provides a mathematically based framework for symbolic evaluation and automatic deduction [100]. The basic evaluation mechanism is similar to backward chaining used in rule based languages for Artificial Intelligence applications, but it is more general and powerful, being based on unification (as opposed to pattern matching, used for instance in production systems) of complex data structures, *i.e.*, logic terms.

From the perspective of a software engineer, there are two main uses of Prolog and its modern implementations: as a declarative, executable specification notation, and as a powerful system programming environment, able to support explorative programming, rapid prototyping, and even multiparadigm system development.

2.1 Prolog for the software specifier

First order logic is widely accepted as a general-purpose notation and programming logic for specification purposes. Prolog uses a subset of first-order logic that can be used as an executable specification language: specifications written in Prolog can even be queried, looking for incorrect or unintended behaviors that imply that the specification itself should be revised.

In short, a specifier can easily use Prolog as notation for specifying requirements; as tool for rapid prototyping of (partial) specifications, as animation language of more abstract specification notations, and as integrator of multiparadigm, hybrid specifications, *i.e.*, written with several different notations.

2.1.1 Prolog as specification notation

A logic language like Prolog is a good specification language because it is a formal language that is expressive, easy to use, and above all executable.

Prolog is actually first order logic in special syntax. First order logic is widely recognized as a basic, universal specification formalism. Even when the specifier writes his initial document in natural language, the use of logic can help in constraining and clarifying the statements: in [121] Prolog is used to parse and process specification statements written in plain English.

However, a formal specification language should exhibit a clear syntax and formal semantics, *and* a programming logic able to support reasoning on specifications [199]. Being based on logic, Prolog allows both writing declarative specifications and reasoning about them [101, 102, 103, 52].

Example:

A classic example of Prolog specification is the following [102]:

```
sort(X,Y):- permutation(X,Y), ordered(Y).
```

This example specifies the *sort* relationship: “argument Y is a sorted version of argument X if Y is a permutation of X and Y is ordered”. The specifier should now develop the specifications of relationships *permutation* and *ordered*.

Admittedly, from the point of view of efficiency, the specification of sorting given above is not satisfactory, being based on the generate-and-test strategy. However, such a specification is a good starting point for developing more efficient programs that exhibit the same abstract properties of the specification. The book [52] is entirely devoted to the development of a formalized programming method that includes a logic to reason on Prolog programs. □

An important aspect of any formal method is the support offered for refinement of specifications, *i.e.*, for obtaining an implementation which correctly implements the specification. For pure logic programming, a notion of refinement has been recently developed [62], that formally defines the concept of logic program that implements a logic specification. However, much work has to be done in this respect to improve the usefulness of pure logic programming for designers.

The use of logic programming as a formal method, *i.e.*, as a language supported by a methodology and a programming logic useful to prove properties of the specification documents, is discussed in [50, 61]. Unfortunately, to our knowledge the only practical application of such a method was the specification of the Prolog language standard; several other experiments with more realistic specifications are needed to validate the method.

It may seem strange to see the same formalism used as both a programming language and a specification language: the two classes appear to be disjoint, because they target different software lifecycle phases. We believe that the difference between a specification language and a programming language is mostly a matter of level of abstraction, cleaner mathematical basis, and performance. Pure logic languages belong to both language classes because of the selected subset of logic that offers the specifier both a declarative and a procedural view: since the Prolog interpreter is actually a theorem prover, programs can be executed in some unusual ways.

For instance, a feature of pure logic programming that is very useful for specification purposes is *invertibility*. A Prolog goal is invertible because it represents a relationship: for instance, the predicate *sort(X,Y)* can be invoked with either argument bound: the result is a value for the other argument: if Y is unbound and X is bound to a list, Y will be its sorted permutation; conversely, if X is unbound and if Y is bound to a sorted list, the interpreter will generate for X all possible permutations of Y.

Not all Prolog programs are invertible; they are not invertible if they include non invertible extra-logical predicates (*e.g.*, arithmetic ones are not invertible). An invertible specification can be used for both generating and recognizing possible sequences of observable actions [99]. This feature has been used extensively in the activity of protocol specification, verification, and evaluation [32, 191]. It is also being used for reverse engineering purposes, for instance to build a decompiler, namely a tool which transforms object code back to source code (see Sect.4.1.1).

From the point of view of a specifier, the most evident drawback of Prolog seen as specification notation is the lack of a type system and structuring mechanisms needed to handle real-life specifications. In fact, data structures are not explicitly declared in Prolog, in contrast to other specification languages, such as Z or VDM. These notations provide tools to check some static semantics properties of formal documents, properties that cannot be easily checked on specifications written in Prolog. Most advanced Prolog implementations offer some form of modularity, but the lack of standardization of such constructs is an obstacle to their effective use.

There have been several attempts to introduce new logic languages offering constructs for declaring abstract data types, generic modules, and overloading; the interested reader will find a discussion of theoretical problems introduced by these concepts in [73].

2.1.2 Prolog as support for rapid prototyping

An executable specification can be considered a prototype; indeed, historically the first applications of logic programming in software engineering were for rapid prototyping of applications [43, 115, 68, 160].

Given a Prolog specification, the execution of the program is similar to the proof of a theorem; however, one relevant difference is the possibility of having nondeterministic search-based computations. The abstraction level of a declarative specification allows one to avoid algorithmic control details; the mechanism of *pattern-directed procedure call* allows a specification to be broken into small, mutually independent, pieces of knowledge avoiding the hierarchical nesting of conditionals typical of algorithmic specifications. This makes easier understanding and maintaining specifications, even when working with incomplete specifications [82].

Logic specifications are also useful for verifying programs written in conventional languages: a logic language like Prolog can be used either alone [8] or in combination with another specification language, *e.g.*, Anna [119], to support the formal verification of modules written in an imperative language, *e.g.*, Ada. The combination of Prolog with Anna is the approach followed in PLEASE, the specification language of the ENCOMPASS environment [184, 185, 186].

PLEASE supports ADA program development and rapid prototyping by incremental refinement of declarative specifications. PLEASE specifications are directly translated in Prolog rules which can be called by Ada modules. In other words, a software system is made partly by full ADA modules and partly by modules whose specification is given in PLEASE and executed by a Prolog engine.

Example:

The following is a PLEASE specification of a sorting module [186].

```
with NATURAL_LIST_PKG : use NATURAL_LIST_PKG
package SORT_PKG is
  --: predicate PERM(L1,L2 : in out NATURAL_LIST) is true if
  --:   FRONT, BACK : NATURAL_LIST ;
  --: begin
  --:   L1 = [] and L2 = []
  --:   or
  --:   L1 = FRONT || cons(hd(L2),BACK) and
  --:       PERM(FRONT || BACK, tl(L2))
  --: end;

  --: predicate SORTED( L : in out NATURAL_LIST ) is true if
  --: begin
  --:   L= []
  --:   or
  --:   tl(L) = []
  --:   or
  --:   hd(L) <= hd(tl(L)) and SORTED(tl(L))
  --: end ;
  procedure SORT(INPUT : in NATURAL_LIST ; output : out NATURAL_LIST);
  --: where in(true),
  --:       out(PERM(INPUT,OUTPUT) and SORTED(OUTPUT));
end SORT_PKG ;
```

Syntactically, PLEASE specifications follow the Anna approach: they are comments. In this example a package is being specified, that includes a SORT procedure. SORT's body is defined in terms of PERM and SORTED, which in turn are defined by a special syntax easily transformed in Prolog code. Such a Prolog code directly executes the PLEASE specifications, and it is interfaced with the Ada support system to allow mixed execution of real Ada code with modules (partially) specified with PLEASE: this is a good basis for rapid prototyping. The special syntax used in the PLEASE specification allows to deal with Ada strong typing, so it may seem strange to Prolog programmers. \square

In PLEASE Prolog is used both as formal notation to specify modules, and as executable notation to be executed; the choice of a development methodology and the burden of proof of correctness of module specifications is left to the specifier.

2.1.3 Prolog as specification animation system

The intrinsic high level of logic programming can probably be exploited best when the specification is written in a specification notation that is not based on logic. More precisely, this approach consists of *animating* a non-executable specification language using Prolog [40]. This is the case of the systems that implement specifications written with the Entity-Relationship model [104], algebraic notations based on abstract data types [83, 22, 75], Petri Nets [9], DeMarco's Structured Analysis Dataflow Diagrams, [112, 111, 72, 147, 56, 189], Z [177, 53, 97, 95, 54, 57, 197, 125, 179], CSP/LOTOS [99], VDM [17], CCS/SMoLCS [71], and domain-specific specification languages [47, 139, 114, 148, 188].

Animation gives a way of "querying" or at least executing the specification, increasing the software engineer's confidence in the document he develops. We describe one example for all: the animation of formal specifications written in the Z notation [175]. Z is especially suited for being animated by Prolog, because it is a formal notation based on first order logic and set theory.

A first systematic approach to Z animation was developed by Knott and Krause in the SuZan project [96]. They studied the feasibility of Z animation, producing a library of Prolog predicates implementing standard Z constructs. They also developed a basic strategy for manual translation of Z to Prolog. The approach they suggested is called *generate and test* because a Z specification is translated to a Prolog program as follows.

A Z specification is a collection of schemata. For each Z schema a Prolog predicate is built which generates all the possible states of variables defined by the first part of the schema, called signature: it contains declarations of typed variables. The translation takes two steps. First, the signature is used to bind the declared variables according to their type; a library of Prolog predicates can be used for this purpose. The second step consists of writing the predicates corresponding to the second half of the schema, which are used to test for solutions. After the translation, it is possible to invoke the schema as a goal which tests the values generated by the signature part.

This approach can handle any specification document written in Z, but is completely impractical. In fact, there is a problem of combinatorial explosion because typically the output variables can be instantiated to candidate solutions only by a complete enumeration of values of types in the schema signature. Often no satisfactory answer will be obtained by a query, either because there are infinite answers, or because the generate and test loop takes too much time.

A different approach consists of transforming a specification document to an "executable" form, so that it is simple to obtain an "equivalent" Prolog program. For instance, in [54] the schemata are presented into a form such that translation to Prolog is immediate. The basic idea is to write "procedural" specifications directly in Z, *i.e.*, the Z schemata are written in a form suitable to be immediately translated into procedures of the target language. This amounts to manipulating the specifications to obtain a prototype of the specification. Signatures are not translated in Prolog, and the goal is only to obtain a program that implements the specification. In this way an operational semantics written in Prolog is given to the Z specification [179]. The specifier must be aware of the declarative and procedural semantics of both Z and Prolog, because a poorly chosen order of predicates or an unconstrained i/o usage of variables can produce programs which loop forever. Moreover, not all Z predicates have a Prolog counterpart, for instance universally quantified statements about output

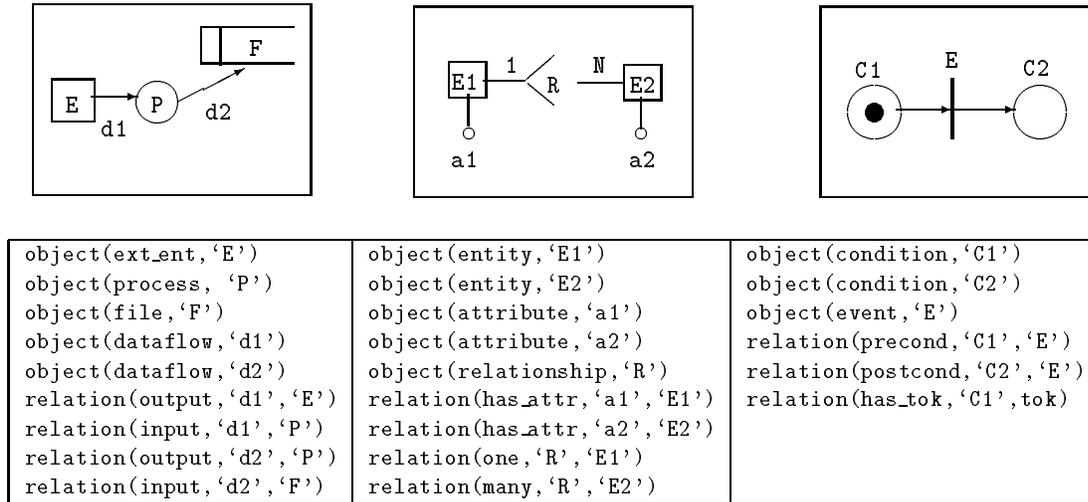


Figure 1: Integrating three different specification languages in PRISMA

variables or ranging on infinite domains have no translation.

There is a field in which Prolog has been largely used for animating and testing specifications: communication protocols [126, 133, 170, 191]. An example is the Mockingbird system, discussed in Sect.3.1.

2.1.4 Prolog as notational integrator

A problem in the specification phase is that possibly several notations are used for describing the product to be developed. In [143, 144, 129] Prolog has been used as an integration strategy among different specification formalisms.

The PRISMA system [143, 144] supports multiple viewpoints in a single specification document. The viewpoints correspond to different specification formalisms; the system is able to integrate them.

Example:

Fig.1 shows three simple examples, respectively a DataFlow Diagram, an Entity-Relationship Diagram, and a Petri Net, and their corresponding Prolog counterparts. For each specification language, a number of predefined basic Prolog facts are used for building a unified knowledge base.

Once the knowledge base has been built, it can be queried. There a number of predefined semantic queries which the specifier can use to query the specification, for instance:

```
has_no_inputs(Proc):-
  object(process,Proc),
  findall(Y,relation(input,Y,Proc),Inputs),
  empty(Inputs).
```

Such a rule checks that a process has no input dataflows in a DFD. □

An approach similar to PRISMA is described in Fig. 2, where the method suggested in [129] to build a unified knowledge base for storing “animatable” requirement specifications is shown.

The idea consists of mapping different specification formalisms on sets of logic facts and rules. The resulting knowledge base is then used to automatically generate test cases for the system being built.

Prolog turns out to be an expressive and flexible specification language mostly because of unification; it is in fact unification which deals with existentially quantified logical variables and provides the specification executor with a simple theorem proving capability.

Unfortunately, there is no agreement on a single general-purpose logic based specification paradigm; usually, different languages corresponding to different specification paradigms are combined into a single

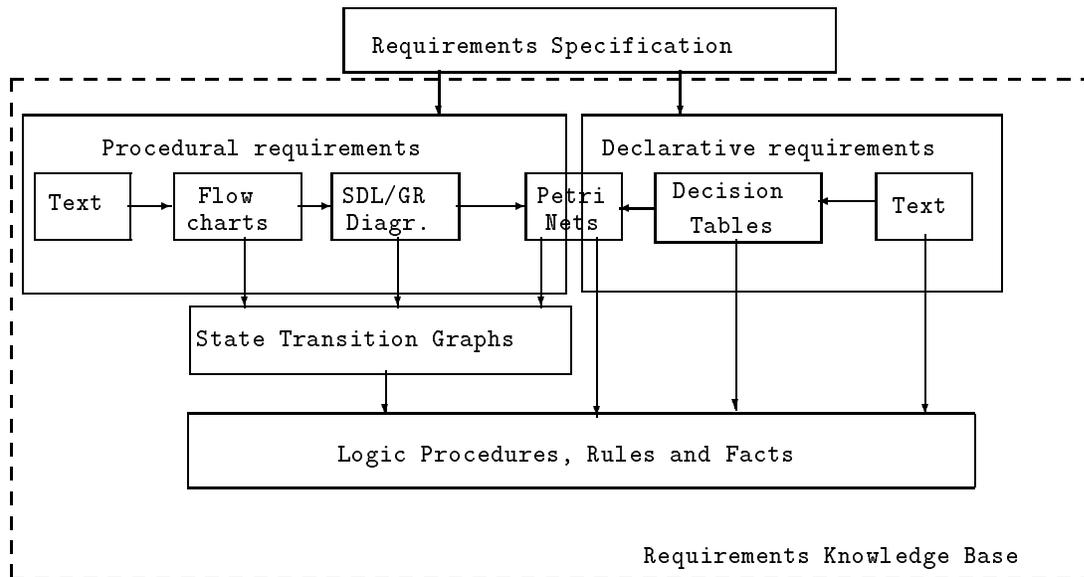


Figure 2: Integration of different specification formalisms according to Mekly and Yuhas

coherent language which is then able to handle each subproblem with the most adequate formalism. An example is the integration of Horn clause logic and equational theories in a number of logic-functional languages: *e.g.*, Eqlg [73], KLEAF [70], Babel [138], etc.

An approach to integrating different specification notations in a single language framework is *Constraint Logic Programming* (CLP), a general language integration mechanism built around Horn clause logic and consistent constraints [86, 87]. The CLP mechanisms enhance Prolog's capabilities for rapid prototyping, mainly because arithmetic, negation, and type recasting are managed in a more elegant way [161]. Constraints are also a powerful (declarative) data abstraction mechanism. The CLP paradigm is shortly discussed in Section 3.1.

2.2 Prolog for the software designer

Usually the main concern in the software design activity is the description of an adequate modular architecture keeping under control inter-module properties like cohesion and coupling. Ideally, a designer would like to have notational and tool support for easily expressing software architectures which match a given specification, for controlling their (re)configuration, and for measuring and possibly improving the quality of the related design documents. Interest in software architectures is arising: a recent work on the use of a logic notation for formalizing them is [118]. Such a paper contains a formal model based on logic for dealing with evolving software architectures, namely for configuration and version management. Instead, in [45] Prolog is actually used as tool for manipulating software architectures by pattern matching.

Also arising is the interest in the quality of design documents: a number of software quality measures can be defined by set of rules which analyze and evaluate program structure. A typical measure important for design is module cohesion: a rule-based, logic approach to measuring module cohesion is described in [109].

In industrial practice Prolog is used in application domains where explorative programming is the main design method. It is easy to describe flexible software architectures using logic programming. There are some classic software architectures that are very easy to deal with in Prolog:

- interpreter: it is normally very simple to implement a prototype, let us say of an editor, as a set of

rules able to parse and evaluate user commands [98]. This approach produces the most inefficient systems: there is an additional interpretation layer causing a significant overhead. However, it is especially useful for designers who practice explorative programming [76, 48].

- parser/compiler: Prolog can be used to translate programs written for a given abstract machine into the machine language of another abstract machine; *e.g.*, [106]. Most Prolog systems offer a special syntax called Definite Clause Grammars to deal with this task.
- toolkit environment: modern Prolog systems are enriched by libraries of standard or useful predicates that can be composed to cheaply build parts of the system to be designed; *e.g.*, [127].
- data/rule base: most Prolog implementations allow to save the knowledge base. In this way it is possible to use a Prolog process as a cheap database system. For instance, in [203] a complex software system was described which included a database component written in Prolog and interfaced with components written in other languages.

All these techniques rely upon extralogical features of Prolog. First order logical systems do not allow the direct manipulation of their formulas and in several cases, for instance when they are used to build a programming environment, they must be extended with some metalevel abilities, by defining a representation of formulas in terms of data structures. A technique called *metaprogramming*, typical of symbolic languages like LISP and Prolog, has been widely used to introduce new language features useful to implement special inference engines [127]. Metaprograms are programs that can manipulate other programs as data, using predefined (built-in) predicates like `call`, `clause`, `assert`, `retract`, etc. (for a comprehensive list and a discussion on their use for metaprogramming, see [180]).

Example:

It is easy to write a metaprogram that interprets Prolog programs and records their proofs. The following example is taken by [140]. We represent the program to be interpreted as a set of clauses and the goal as a list $[G_1, \dots, G_n]$, where every G_i is either a `system(Goal)` or a `user(Goal)`; system goals correspond to built-in Prolog predicates.

```
% a Prolog metainterpreter which records the proof of a goal

prove([A|B], (Pa,Pb)):-
    prove(A,Pa), prove(B,Pb).
prove(G,system(G)):-
    system(G), call(G). % call is a built-in that invokes the interpreter
prove(G,(A:-Pb)):-
    user(G), clause(G,Body), prove(Body,Pb). % clause is a built-in that reads a rule
```

We can invoke such an interpreter with a goal like `prove(G,P)`. The rules for `prove` say: “if the goal G is a list, decompose it and prove separately the head A and the tail B ; if the goal is a system goal, call the interpreter; if it is a user goal, the built-in `clause` predicate reads the representation of a rule whose head unifies with G and substitutes to G the rule $Body$ ”. □

Several programming tools, ranging from more detailed metacircular interpreters (*i.e.*, interpreters which specify more in detail the run-time behavior, for instance expliciting how data structures are represented and dealt with [106]) to complete programming environments [130], can then be defined as metaprograms. Writing a tool as a metainterpreter is a rather easy task: for instance, the metainterpreter given above can be easily modified to instrument a program to print information useful for debugging, *e.g.*, to pretty print the run-time behavior (*i.e.*, the proof) of a goal, or to measure some property of the execution, *e.g.*, the number of inferences necessary to obtain a result [76, 48].

Example:

It is easy to write a metaprogram which pretty prints the execution generated by the metainterpreter in the above example [140].

```
% a metaprogram which pretty prints the proof of a goal

printProof(X):- printP(X,0),nl,nl.

printP(system(true),_):- write(' fact').
printP(system(A),D):- nl,tab(D),write(A),write(' system').
printP((A:-Pb),D):- nl, tab(D),write(A),write(' :-'),D1 is D+3,printP(Pb,D1).
printP((Pa,Pb),D):- printP(Pa,D),printP(Pb,D).
printP(Pa,D):- nl, tab(D), write(Pa).
```

We can invoke such an interpreter with a goal like `?- prove(G,P),printProof(P)`. obtaining a pretty printing of the proof. \square

The performance of logic metainterpreters is acceptable since the computationally complex tasks, *i.e.*, unification and backtracking, are typically mapped onto the corresponding tasks of the implementation language. Moreover, the efficiency of metainterpreted programs can be improved by using *partial evaluation*, a form of compilation. Intuitively, to partially evaluate a program means the opposite of generalizing it. When a very general program, for instance an interpreter, is partially evaluated, we obtain a sort of compiler, less general but more efficient than the interpreter [162, 183]. The combination of metaprogramming and partial evaluation makes simple and effective the implementation in Prolog of new language mechanisms and programming environment tools [98].

If the result of partial evaluation is still inefficient, it is always possible to use the metaprogram as the specification of a new implementation in a more efficient language: this technique was used in the design of the compiler for the real-time programming language Erlang [4]; the authors report an improvement of more than 70 times in speed for the compiler obtained in this way, with respect to the metainterpreter.

The flexibility and power of *metaprogramming* as a system programming and rapid prototyping technique has been proved in several independent projects, that have used it mainly in the design and implementation of enhanced programming environments for logic languages *e.g.*, [106, 98, 39]. Metaprogramming is particularly important in explorative programming, because the development and testing of a prototype must be supported by tools similar to those used for the development of artificial intelligence systems. This requires good interactive integrated editing and program analysis tools.

Even if programming tools can be implemented in any language, the LISP and Prolog experiences show that using a language to define its own tools is, whenever possible, the best choice, since the set of tools can easily be extended to prototype new language mechanisms and environments.

3 Beyond Prolog: Constraints, Concurrency, Objects

A software engineer should not restrict his perspective to Prolog only: there are now several logic programming systems with novel features not found in Prolog systems that are very useful for software engineering applications.

Some important properties of software systems cannot be specified by a sequential language like Prolog. For instance, the exploitation of parallel and distributed architectures is impossible: the language is sequential. Moreover, the main computing mechanism in Prolog, *i.e.*, unification, is too limited and ineffective to deal with systems based on constraint solvers. Finally, the lack of modularity mechanisms in Prolog makes difficult and expensive the design and construction of large programs. Thus, new languages have been proposed to deal with these classes of applications, *i.e.*, constraint logic languages, concurrent logic languages, and object-oriented logic languages.

Unfortunately, currently we still do not know if these new languages will eventually be provided with implementations of efficiency comparable to those of Prolog, availability in different operating environments, and good portability.

3.1 Constraint Logic Programming

The execution of a pure logic program is based on the unification algorithm, which, given an equation $t_1 = t_2$, performs two tasks: it decides if the equation has solutions, and it simplifies the equation, transforming it to the solved form (a set of variable bindings). Normal logic programs compute on symbolic data like atoms or lists by accumulating sets of solvable equations on the domains of atom and lists; however, these data domains can be replaced by other domains, like integers or reals. Computing can then be accomplished by accumulating sets of solvable equations over the new domains. When applied to logic programming, this method is called constraint logic programming (CLP): a good survey on this subject is [86].

Constraint systems allow to make inferences on a data domain without knowing the actual representation of data and the underlying implementation of the basic operations. There is no difference between execution, symbolic execution, and theorem proving. This feature is at least potentially interesting for specification applications.

Syntactically, constraints can be atoms corresponding to procedure calls in different languages $\mathcal{L}_1, \dots, \mathcal{L}_n$. The constraint solver is then a set of proof procedures for each \mathcal{L}_i . One inference step in CLP combines several inferences in different languages.

For each constraint system we need a consistency algorithm. Pure logic programs compute answer substitutions, *i.e.*, explicit representations of sets of solutions. Constraint logic programs compute sets of solutions implicitly represented by constraints.

The CLP scheme maintains all the nice semantic properties of standard logic programs. It includes a canonical domain (a many sorted algebraic structure) to compute; there is a least fixpoint semantics and a least model, soundness and completeness for successful derivations can be proved, and soundness and completeness of negation as finite failure can be proved. Moreover, all these properties are inherited by any “extension” which can be formalized as an instance of CLP.

Example:

A constraint logic language has been used in the Mockingbird project, where the specification of a message protocol is compiled into a constraint-based logic program used for both generating and validating test data [74].

Mockingbird allows one to specify a message protocol as two components: a grammar over the domain of bit strings (for syntax) and a constraint system (for semantics constraints). For instance, in the following table a toy datagram protocol is described by a set of BNF rules; to each rule a number of equalities and/or inequalities are associated. In the second rule s is a bit string.

<code>datagram → header,octects.</code>	<code>header.length = octects.length + 8</code>
<code>header → source,destination,length,checksum.</code>	<code>0 < source</code> <code>0 < destination</code> <code>8 ≤ length ≤ 65535</code> <code>checksum = source + destination + length</code>
<code>source → s.</code>	<code>s.width = 16</code> <code>source = s</code>

For any Mockingbird specification there is an equivalent CLP program. What follows is the CLP counterpart of a simple datagram communication protocol.

```
% partial specification of a toy datagram using constraints
datagram(In,Out):-
    Length0 = Length1+8,
    header(In,Out1,Length0),
```

```

octets(Out1,Out,Length1).
header(In,Out,Length):-
  Source > 0, Destination > 0, Length >= 8, Length <=65535,
  Checksum = Source + Destination + Length,
  source(In,Out1,Source),
  destination(Out1,Out2,Destination),
  length(Out2,Out3,Length),
  checksum(Out3,Out,Checksum).
source(In,Out,Source):-
  Width = 16, Source = S,
  s(In,Out,S,Width).

```

These clauses are obtained by compilation of the above specification. We can query the program to parse a given bit string; each rule body first generates a number of constraints then applies the usual backward chaining ruled by backtracking to actually recognize the input bit string. The clauses can be queried in two ways: either to validate a message, *i.e.*, to prove that it satisfies the constraints, or to generate sets of valid messages useful for testing a real implementation of the protocol. \square

Operationally CLP is very similar to standard logic programming: clauses are selected and reduced; but while a Prolog goal execution produces a set of variable bindings, a CLP goal produces a system of constraints. The order of evaluation of numeric predicates is irrelevant, because solubility is a property of the whole constraint system.

3.2 Concurrent Logic Programming

Reactive systems are an important class of software systems, including operating systems, real-time systems, interactive systems, etc. The construction of such systems requires the use of both ad hoc specification languages, to be sure to capture all the intended properties required by the users and the environment in which they are embedded, and ad hoc programming languages, because usually such systems are distributed and have a complex i/o behavior.

Can the logic programming paradigm be adapted to the specification of *reactive systems*? Reactive systems are characterized by their intrinsic concurrency and ability to react to messages coming from the environment. The intended semantics of a reactive program is a sequence of stimulus and reaction events meaningful even in the case of non-terminating or failing computations.

Even if they are more complex than sequential logic languages, concurrent logic languages have similar mechanisms; in fact, the mechanisms for handling concurrency are based on the typical features of logic programming: a good survey is [166].

The simulation of the state of a process uses partially specified data structures; synchronization among processes is achieved by modifying unification or by constraint accumulation in a shared space of constraints; control of nondeterminism is simplified by committing to only one nondeterministic computation among all those that are possible (*don't care* non-determinism).

A *program* in a concurrent logic programming language is a set of logic processes, each one described by a set of clauses of the form:

$$p(X_1, \dots, X_n) : - ask_1, \dots, ask_p : tell_1, \dots, tell_q | b_1, \dots, b_m$$

where

- the ask_i 's (*ask guard*) and the $tell_j$'s (*tell guard*) are synchronization constraints: an *ask* constraint simply checks that some constraint is true with respect to the current set of valid constraints, whereas a *tell* constraint adds a new constraint to the set of valid constraints;
- $|$ is the *commit* operator;
- the b_k 's (*body*) are invocations of other logic processes;

The ask guard checks for constraints that must be true before using the corresponding rule; the tell guard asserts new constraints; the body spawns new parallel processes. Guard evaluation has to be thought of as an extension of the unification with the clause head; when the unification and guard evaluation succeeds for a clause, the computation can commit to that specific clause and no backtracking is allowed afterwards.

There are several concurrent logic dialects, that differ in syntactic form and semantic details of the rules [166]. One of the most known is Concurrent Prolog [167].

Example:

Concurrent Prolog was introduced as a concurrent logic language useful to write parallel programs with a dataflow-like style. The following is an example of a logic process that defines a (shared) counter.

```
% a counter class declaration
counter(Input,State) :-
    Input = [clear|I] |           % msg clear resets the counter
    counter(I?, 0).              % recursive call to make object persistent
counter(Input,State) :-
    Input = [up|I] |             % msg up increases the counter
    plus(State,1,NewState),      % NewState=State+1
    counter(Input?,NewState).
counter(Input,State) :-
    Input = [down|I] |           % msg down decreases the counter
    plus(NewState,1,State),      % NewState=State-1
    counter(Input?,NewState).
counter(Input,State) :-
    Input = [show(State)|I] |    % msg show is incomplete
    counter(Input?,State).
counter([],State).              % the input stream has been closed, counter terminates
```

A **counter** process has two arguments: an input stream and its internal state; its behavior is defined by five rules. Four of them handle the possible incoming messages; the last one terminates the counter. The first rule resets the counter; the second and the third rules respectively increment and decrement the counter; the fourth rule asks for the state of the counter. The last rule terminates the object when its input stream is closed.

In the fourth clause of **counter** the message **show(State)** contains an unbound variable. This is an example of incomplete message, a technique that allows a client process to obtain from **counter** the answer to a message using part of the message itself. Typically, after sending to **counter** an incomplete message (a term containing an uninstantiated variable), a client process suspends itself. It will be resumed when the answer is ready (that is, when the variable in the incomplete message becomes instantiated).

The following goal defines a system that includes a **counter** process:

```
terminal(StandardInput),use_counter(StandardInput?,Channel), counter(Channel?,0).
```

This goal starts three logic processes: a **terminal** process, a **use_counter** process that reads the *StandardInput* stream and produces messages on the *Channel*, and a **counter** process that reads messages on the *Channel*. The initial value for **counter** is 0.

Fig. 3 show a graph that depicts the relationship among the three logic processes created in the above goal (arrows are streams, circles are processes). □

3.3 Object-oriented Logic Programming

Apart from constraints and concurrency, another aspect that has been considered by researchers is the embedding in logic programming of different programming constructs and paradigms. The combination

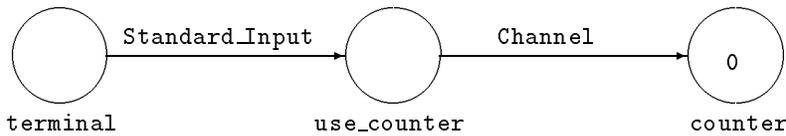


Figure 3: A process graph corresponding to a CP goal

of the object-oriented paradigm with logic programming has been described in several papers; a good survey is [44].

The most appealing features of logic programming are found in the elegance of its semantic characterization and in the declarativeness of its computational model. However, in spite of its declarativeness, pure logic programming does not scale when it is used to design real-life applications. The basic abstraction unit, *i.e.*, the relation, is too fine grained to support the development and maintenance of large specifications and programs [122].

Modularity, inheritance, and reusability are classic object oriented concepts which simplify the development of large applications. At the heart of object-oriented programming is the notion of object with its associated set of methods.

From a logical point of view, an object, the basic abstraction unit, has a natural interpretation as a logic theory: an object is simply a collection of axioms which describe what is true about the object itself. Operationally, clauses defining the object can be used as methods to be invoked.

Example:

A simple object oriented Prolog extension was proposed by Zaniolo [202]. The extension was based on the operators **with** (object declaration), **isa** (inheritance), and “:” (method invocation).

```

% logic objects according to Zaniolo [202]
parallelogram(Base,Height) with [(area(A):- A is Base*Height),(what_is_it(a_parallelogram))].
rectangle(Base,Height) isa parallelogram(Base,Height).
rhombic(Base,Height) isa parallelogram(Base,Height):- Height <= Base.
square(L) isa rectangle(L,L).

```

This program fragment declares an object class **parallelogram** and three classes that inherit part of their behavior from it. Such a program can be queried as follows:

```
?- square(5):area(L).
```

Such a goal, that invokes a method of the class **parallelogram**, succeeds with answer **L=25**. □

The design of a logic language extended to incorporate modularity and inheritance must take into account different levels of integration. At the procedural level, a new inference system has to be defined, combining the new mechanisms with the deductive process of resolution [134]. At the declarative level, inheritance must be characterized in terms of the standard notions of satisfiability and truth found in classical logic; moreover, the designer must solve the problem of capturing the compositional properties inherent in the incremental approach to software development entailed by inheritance. For a detailed discussion on these issues, see [19].

There are many advantages in combining logic and object-oriented programming, because the resulting languages overcome most limitations of both approaches, taken singularly. For instance, logic programming benefits because an object-oriented style helps in the comprehension of programs including entities that encapsulate a notion of state, that abstractly is alien to the logic programming paradigm [195]. Moreover, the lack of structuring mechanisms for programming-in-the-large typical of Prolog can be removed using a class construct, that is typical of object-oriented languages. A difficulty here is that a notion of class or module hierarchy must be carefully defined in a logic programming framework [131].

Conversely, some logic programming features are very interesting in an object-oriented framework: for instance, in conventional object-oriented languages it is difficult to express relationships between classes, except by inheritance. Another issue largely investigated is the use of the logical variable for enhanced communication mechanisms in a concurrent object-oriented framework [44].

The logical variable in combination with unification offers a simple mechanism able to deal with several different programming concepts, like assignment, value testing, complex data construction, and several forms of communication. However, the main contribution of logic programming from the point of view of object-oriented programming is the availability of clearly defined mechanisms to specify concurrency, synchronization, and i/o behavior.

3.4 Discussion

Which kind of applications will benefit from the new research trends of logic programming? Keeping the main features of conventional logic languages, *i.e.*, declarativeness and flexibility, the new logic language systems are being used for a new range of applications, too complex to be written in standard Prolog.

Constraint logic languages are being applied in a large range of applications, like operations research applications, circuit design, macro economics modeling, investment planning, etc. [55, 161]. Their application in software engineering projects is still limited, with the notable exception of tools for compiler testing [74].

Concurrent logic languages are comparably newer than Prolog, so they have been less used in software engineering research and practice. They have been largely used to specify and prototype some programming systems and tools, like a parallel parser [187], an operating system [171, 65], a parallel programming environment [64, 63], and a window system [91]. They are also usable as animation languages: an example of parallel animation of dataflow diagrams using a concurrent logic language is given in [176].

They have been proposed as specification languages for concurrent and distributed systems [182, 69, 58, 2]. More specifically, the use of Concurrent Prolog as a specification formalism for hardware and software systems was investigated in [182, 107, 2, 108]. However, the lack of a clear semantics and a programming logic limits their usefulness.

The combination of object-oriented and logic programming is even more popular than the preceding approaches. This kind of language is often used to design and prototyping programming environments and tools [131, 140]. The extensibility and modularity typical of the object-oriented paradigm integrate the declarativeness and flexibility of the logic paradigm, helping the designers in mastering the complexity of such environments.

4 Applications of Logic Programming in Software Engineering

Several independent experiences, both in the universities and in industries, have shown that logic programming has a great potential for reducing the cost of software development. In this section we will summarize a number of projects that successfully used logic programming as the basis for developing software development environments and tools.

A problem that we encountered while we were collecting the references for this paper is that there are several researchers who have used logic programming languages or systems to design or implement software engineering tools, but with varying approaches. For instance, some applications use Prolog as a rule-based language; others use it as a relational data modeling and query language; others as an executable specification language. Sometimes logic languages other than Prolog were used as well.

For our convenience, we have partitioned the projects we survey in three categories: projects and tools oriented to programming-in-the-small (one programmer - one module), including tools like compilers, structure editors and debuggers; projects and tools oriented to programming-in-the-large (one programmer - several modules), including tools for configuration management, module reuse, inter-module testing, and automated test-case generation; projects and tools oriented to programming-in-the-many

(several programmers - several modules), including process modeling languages and rule-based process-centered environments actively coordinating and supporting the software development process.

4.1 Programming in-the-small

We classify here tools and environments designed to be used by one programmer working on one software document (*e.g.*, a program module): compilers, structure editors, symbolic debuggers, and testing tools.

4.1.1 Compilers

Compiler-writing tools are possibly the best known and most powerful software engineering tools. However, they are often complex, being composed by several subsystems (*e.g.*, scanner generators, parser generators, type checkers, code generators, etc.) so that they need expert practitioners. Using logic programming, a single formal framework can be used for all components.

Special features offered by logic programming languages for the development of compilers have been exploited quite early in the history of this technology. In fact, already in 1975 a seminal paper by A.Colmerauer on Metamorphosis Grammar [37] suggested many of the successive developments in compiler writing using logic programming.

In fact, Warren convincingly proved the power of Prolog in the activity of compiler construction [194]. The main idea is to specify a compiler by logic rules that state how each construct of the source language is converted into a sequence of object code instructions.

Example:

The following Prolog rule specifies the top level of a compiler [194]:

```
% specifying a compiler of a source into object code with absolute addresses
compile(Program, (Code; instr(halt,0); data_block(L)):-
    encode_statement(Program,Dictionary,Code),
    assemble(Code,1,N0),
    N1 is N0+1,
    allocate(Dictionary,N1,N),
    L is N - N1.
```

The compiler is defined by a rule that states that the result of compiling a program is a sequence, separated by semicolons, including some Code representing object code instructions, followed by a **halt** instruction, and finally by a data block that is the symbol table that stores variables used in the program. Predicate **encode_statement** defines the actual code generation, that is associated to the generation of a symbol table stored in Dictionary. Predicate **assemble** computes the address of instructions and returns the length N0 of generated object code, that is used by predicate **allocate** to put in place the symbols contained in the Dictionary and to compute the overall memory size L required by the dictionary itself. □

After Warren's pioneering work, the paper [36] reviewed several techniques for implementing parsers and compilers in Prolog, providing a complete framework for such an activity.

In the case of compiler writing, it is clear that the use of unification and nondeterminism helps to write concise programs, that are easy to read, simple to debug, and even suitable to be proven correct capitalizing on the formal foundations of the language, *i.e.*, first order logic.

In fact, the concept of formal specification based on logic programming has been exploited in [24, 178] to formally develop a compiler and its correctness proof. In [24] a compiler is specified by a set of rules, each describing how a construct in the source language is to be translated in the machine language; in a second phase a refinement algebra for the source language is used to develop a correctness proof for the compiler. In another work [23] is also showed how this approach can be used for reverse engineering, insofar as a compiler specified by a program written in pure Prolog is easily used as a *decompiler*, that is a tool can obtain the source program corresponding to a given object code. In [178] a compiler is obtained defining in Prolog the denotational semantics of the language to be compiled; the denotational semantics

is represented by a set of rules written in the formalism of Definite Clause Grammars (DCG). These are a generalization of context-free grammars and are executable, because they are simply a notational variant of Prolog programs [180].

It is honest to say that most compilers written in Prolog compile either Prolog itself or logic languages. However, Prolog can also be used to rapidly prototype compilers for imperative languages. Its usefulness has been confirmed in [151], where the development in Prolog of a compiler for Edison, a Pascal-like language, is described. According to Paaki, Prolog compares positively as a compiler development tool with other, more traditional approaches based on imperative languages. Paaki also suggested to extend Prolog with specific mechanisms useful for a compiler designer [150]. Finally, he surveyed the use of logic programming technology in support of attribute grammars and parsing tools in a recent paper [152].

At least one commercial compiler, namely for the ESTELLE language, has been fully implemented in Prolog [136]. Another language that has been initially implemented by a metainterpreter written in Prolog is Erlang [4]; since Erlang is intended as a real-time programming language, for efficiency reasons it has been subsequently rewritten in C.

4.1.2 Structure editors

The successes obtained during the early experiences in the design of language parsers and translators convinced a number of researchers of the effectiveness of Prolog as a language for rapid prototyping of basic programming tools.

In language-based environments, programs usually are represented by abstract syntax trees which can be analyzed or manipulated by several language-generic tools, like parsers, static semantic analyzers, dataflow analyzers, code generators. The language specific part of most tools is specified by a grammar augmented with informations used by the tools. Such information is usually given in form of attributes to the productions of the grammar.

Several projects have focused on the development of structure editors based on logic programming. A *structure editor* is a tool that uses a program representation based on abstract syntax to incrementally perform a number of semantic analyses on the program being coded.

A logic program has a direct interpretation as set of grammar rewrite rules [51, 78, 158], so that it is easy to specify the abstract syntax and static semantics of a programming language. These works showed how an attributed grammar can be immediately interpreted as a logic program that can be used as a symbolic interpreter.

This is also the basic idea in Centaur [18]. The Centaur system aims to support symbolic manipulation (editing and evaluation) of structured documents. Its kernel is composed of two main components: a syntactic tree editor (called Virtual Tree Processor) that handles syntactic aspects of documents, and a symbolic interpreter (written in Prolog) of executable documents. The symbolic interpreter evaluates semantic language specifications written in Typol, a language apt to define operational semantics by transition systems. Typol programs extend a programming in-the-small environment for a given language with standard tools like debuggers, type checkers, and so on. The Typol interpreter was written in Prolog to take advantage of logic variable unification, a very useful mechanism to simulate evaluation by transition rules. The Prolog evaluation in Centaur consists of solving a set of equations that identify values, types, states, etc.

Another structure editor based on (constraint) logic programming is Pan [10]. More precisely, this is a language-based editing and browsing system. It allows the incremental static analysis of programs whose semantics is specified with a logical constraint grammar. A Pan language description includes declarative information on lexical and syntactic data, context-sensitive constraints that state the static semantics, and user-interface specifications to configure the editing tools according to the need of different users or different tasks.

The heart of the system is a formalism that adapts logic programming for consistency maintenance of a knowledge base associated with the abstract syntax tree of a program using logic constraint grammars. A *logical constraint grammar* (LCG) is a context-free grammar in which symbols and productions are annotated with goals expressed in a logic programming language (instead, DCG mentioned above are

an executable, Prolog-like notation for context-free grammars). The goals in LCG productions express constraints on the nodes of abstract syntax trees generated by the grammar.

Example:

A logical constraint grammar contains the operational description of contextual constraints over the nodes of an abstract syntax tree representing a program. We show two grammar rules and the constraints that are associated to them.

```
% a logical grammar fragment requiring that each name be defined before it is used
< document > → < def >* < use >*
< def > → 'DEF' id
      :- context($$,?Scope),
         string-name($id,?Name),
         not(<visible(?Name,??), (?Scope) >): 'Invalid redeclaration of Name in this scope',
         new-entity(?Entity),
         assert(<type-of(id), ?Entity >),
         assert(<declared(?Name, ?Entity), ?Scope >).
```

The first rule is an unannotated production: no constraints are given. The second rule checks the declaration of an identifier `id` following the keyword `DEF`. In the logic rule, an identifier prefixed by “\$” denotes a node in the abstract syntax tree; “\$\$” denotes the node associated with the goal “currently” being satisfied. The goal associated with the production accesses the context to get the current scope, binds the identifier to `?name` and checks that no other binding occurs in the current scope for such an identifier. If such a check is satisfied, two facts are asserted, which enrich the structure editor knowledge base with data about the new identifier. □

Structure editors are not in wide use in software engineering practice, and those which are based on logic programming are even more limited because they need a lot of computing resources, especially memory. However, they are indeed powerful tools and we believe that they will increase their importance in the future, when more and more documents based on formal syntax of any kind will have to be automatically handled in complex software processes 4.3.2.

4.1.3 Symbolic Debugging and Analysis Tools

Any software engineer knows that debugging and analysis tools are very helpful in his work. Every programmer has a preferite toolkit to manipulate his programs. However, it happens to miss some functionality in the personal toolkit, so many software engineers usually build ad hoc tools for some specific task. This is an activity in which the use of Prolog is very advantageous.

Debugging from the point of view of programming-in-the-small is very well supported in modern Prolog programming environments, that generally offer a full-fledged query language to manipulate the execution of Prolog programs. Recently this flexibility has also been proposed for non-logic languages [59], aiming at building powerful trace query mechanisms for executions represented by stream of events. Even when the environment does not offer a powerful debugging tool, it is possible to develop ad hoc tools by metainterpretation [49].

Prolog is obviously well suited for implementing standard tools and techniques for symbolic debugging. A first work which showed how metaprogramming can be used to build symbolic debuggers is [165], with application to Prolog itself. In [92], Khanna shows how it is possible to build a symbolic interpreter for a generic language building a graph representing the abstract syntax of the program to be verified. The interpreter implements a path testing strategy, namely each path in the program has to be traversed at least once. Multiple logic theories are built, one for each execution path traversed during the symbolic execution. The theories can be queried for debugging purposes.

Example:

Symbolic evaluation of a program is based on a symbolic environment represented by a list of variable-value pairs:

$[(Var1, SymbVal1), (Var2, SymbVal2), \dots, (VarN, SymbValN)]$

Values associated to variables can be either numeric or symbolic.

An interpreter for the language to be evaluated is defined by a set of rules implementing a symbolic operational semantics. The predicate representing the interpreter has the following interface: `symb_eval(Command, OE, NE, OSE, NSE, OCOND, NCOND)`, where `Command` is a program fragment, `OE` and `NE` are the numeric (non-symbolic) environments before and after the evaluation of the `Command`, `OSE` and `NSE` are the symbolic environments before and after the evaluation of the `Command`, `OCOND` and `NCOND` are predicates recording the (symbolic) decisions taken along the path followed to reach the `Command`.

For instance, the following rule implements the evaluation of an assignment in an imperative language.

```
symb_eval([VAR, ':=' , EXP], OE, NE, OSE, NSE, OCOND, NCOND) :-  
  expr(Value, OE, EXP, []),  
  change_env(OE, [VAR, Value], NE),  
  symb_expr(SymbolValue, OSE, EXP, []),  
  change_env(OSE, [VAR, SymbolValue], NSE).
```

That is, to evaluate an assignment `VAR:=EXP` the `EXP` is evaluated both numerically and symbolically, and `VAR` gets a numeric value, if possible, otherways a symbolic value. \square

A number of projects have used logic programming technology for building analysis tools: for instance, Ergo [113] and Focus [157]. Ergo is an integrated set of programming tools. Among these, one is implemented in Prolog to use first order logic for reasoning in semantic domains and for program verification and synthesis. Focus is a programming environment that integrates the logic and the functional paradigms for automatic program derivation.

A set of tools for analyzing small programs is described in [76]. Several software analyzers are described, integrated in an environment able to produce self-instrumenting programs. In [88] is described PQL, a language-independent language useful to query a knowledge base (implemented in Prolog) storing data on programs. The queries implement a wide spectrum of static analysis checks. The WSDW project [48] is also a programming environment based on a Prolog knowledge base that integrates a set of interactive programming tools, offering a shared level of program representation. A program is represented in form of relations which are stored in the knowledge base. Tools based on rewriting rules are able to restructure the program for optimizations or to instrument the program for debugging.

Finally, we want to cite an original approach to the problem of debugging in-the-small. In [67] a rule-based user model is described that is the basis for a system able to learn to provide meaningful on-line diagnostic data that can be used to help and guide the user. The model is based on the formal semantics of a programming language; the sample language used is Prolog itself. Here the use of Prolog was useful because the diagnostic program is a sort of expert system.

4.1.4 Testing

As an alternative to symbolic debugging, testing methods have also been defined using knowledge-based strategies that are easy to implement using logic programming. The basic idea consists of automating the testing process. The execution of very large test data sets is performed by test drivers which can be built by a system expert in testing, thus saving the software engineer the boring and error prone task of developing large test suites.

In [46] *rule-based software testing* was introduced: heuristic testing methods for Ada modules were formalized in a knowledge base written in Prolog. The rules derive from a set of criteria including statement coverage, condition coverage, decision coverage, and multiple-decision coverage. The Prolog program then is used to generate test cases: a test case consists of a set of input values, a set of expected output values, and observed results for one execution of the program to be tested. This approach compares favorably with randomly generated test cases. Other projects are pursuing this approach, notably [79, 128].

Example:

Rule-based software testing is a knowledge-based approach: the software engineer has to formalize his knowledge on testing some specific class of modules. For instance, a typical rule described in [46] is:

IF the program contains a condition and there is a test case for an outcome of the condition, THEN generate test cases by doubling and halving the values of the previous best test case.

This kind of rule allows to “stretch” the testing space of possible input values. □

A more abstract approach consists of matching the execution of a module against his executable specification. This is the case of the project described in [80, 81, 181], which uses Prolog as support for automated module testing. Protest is a tool that tests the implementation of a module written in C. Protest includes two subsystems: the first one tests a C program using test cases containing the expected output, whereas the second compares the behavior of a C program to that of a Prolog specification called the *test oracle*. Test cases are described in terms of traces of module invocations. The tool is able to compare the trace sequence of actual program with the expected trace sequence defined by a Prolog specification of the program itself.

Another Prolog-based testing tool is described in [14]: a specification language is used to define a special testing strategy called “All-du-paths”; the specification allows to associate a logic database to each program module; the tool analyzes each module database and offers an estimate of the minimum number of testing cases necessities for satisfying the testing criterium.

The combination of logic and algebraic languages has been exploited to develop tools for automatic generation of test suites [154, 20, 21, 12, 42, 123]. The idea is that an algebraic specification, *e.g.*, of a data type, can be implemented by a Prolog program which in turn can be queried. The query tries to prove some property of the specification; the answer to the query allows to define a test data set which can be used to test a program implementing the specification.

4.2 Programming-in-the-large

The main goal of an environment for programming-in-the-large is to control and support the development of large software projects, that usually involve several programming modules, possibly to be developed by different programmers. Typical activities that should be supported by an environment for programming-in-the-large are configuration management, version control, and automatic retrieval of data relevant for software development, *e.g.*, software reuse of existing modules.

4.2.1 Configuration Management and Version Control

Several activities concerning configuration management and version control can be coordinated by tools that use rule-based languages, as for instance the **make** utility under Unix. Thus, it is not surprising that Prolog has been used to build knowledge-based tools to support rule-based coordination of software maintenance activities.

For instance, [93] describes **prom**, a tool based on Prolog that extends the capabilities of the **make** utility. **prom** exploits the deductive capabilities offered by logic programming in manipulating a knowledge base describing a large software system. Prolog is used as a database allowing to write makefiles in a declarative way. **prom**’s makefiles consist primarily of Prolog-like terms. This allows to describe relations between components of software systems abstracting from lexical details of file names, etc. , while separating the concerns of target architectures from the relations among software system’s components. This is the key to maintain several variants of software systems for different purposes or target architectures. Typically well done in Prolog is the declarative nature of makefile entries. **prom** allows to declare things specific to a component, a group of components, or even all components. **prom** allows a declarative “high-level” description of software configurations without being fixed to lexical rules as in **make** (*e.g.*, from **.c** to **.o**) where the “.” is crucial for a correct behavior.

A similar approach was also developed by the designers of DERIVE, a deductive database system supporting software configurations [173, 174]. DERIVE uses partial evaluation to produce configurations

described by makefiles and scripts; it also employs abstract interpretation to answer queries about product feasibility.

A knowledge-based tool designed for multiparadigm configuration management is Polygen, a tool used in the Polilyth environment [29]. Polilyth allows the integration of distributed and heterogeneous software components. Polygen is a knowledge-based tool written in Prolog used to generate the software interfaces needed to integrate and transform (for instance, to compile and link) the description of a software configuration into a set of distributed and communicating executable objects. The configuration process is driven by a set of composition rules that define the integration capabilities of the operating environment. Modules can be composed and inserted in different applications without being explicitly modified, thus enhancing their reuse rate.

Example:

Polygen is a tool able to decide the way some software components should be integrated given some integration capabilities typical of an operating environment. Module specifications are given in form of a set of Prolog assertions. A site administrator gives the configuration rules valid to integrate component in a given environment. For instance:

```
% facts about a module
module(client).
language(client,'C').
source(client,'client.c').
include(client,'client.h').
main(client).
% A configuration rule in Polygen
package(Configuration,Target):-
    modules(Configuration, Modules),
    instances(Modules, Instances),
    partition(Instances, [[Configuration,Target,Package]]),
    createpackage(Modules,Instances,[Configuration,Target,Package]).
```

The rule states that to make a package, given a configuration, the modules included by the configuration must be instantiated and then partitioned, *i.e.*, recognized as compatible for integration according to the constraints of a given operating environment.

The knowledge base can be queried as follows:

```
?- package(test,tcip).
```

that means: “is it possible to create a package for configuration `test` using `tcip` environment?”. The final result is a script created by `createpackage` that is then executed by the Unix `make` program, that actually builds the executable codes. Another possible kind of query is `package(test,Env)`, that suggests an environment `Env` that offers the capabilities necessary to configurate a package `test`. □

An approach similar to Polygen, where a logic programming language is used as module interconnection language, is discussed in [11].

A tool developed for helping software engineers in the configuration design phase is described in [38]: a graphic language called GraphLog is able to represent complex software objects and their relationships. The visual representation of a software system can be manipulated aiming at simplifying its structure. The basis for GraphLog is a logic language simpler than Prolog and similar to DataLog, a database logic programming language.

More complex approaches are possible, using a logic programming language to implement non standard logics able to reason with special features of a software system. For instance, in [142] SCAV is described, an access and version control system including a reasoning tool based on temporal logic and realized in Prolog. Temporal reasoning is used to manage the version history of a software project, aiming at minimizing storage requirements of the set of modules created during the development. In [6, 5, 7] a logic database called EDBLOG was used to build systems able to reason about modules properties and configuration strategies of programs written in Ada. The logic database exploits an extension of

Prolog that supports constraints: these are introduced to guarantee the consistency of a project database containing modules with respect to a number of global properties. A makefile utility was built, using the features offered by the constraint language. Similarly, in [172] a logic database supporting the construction and evolution of software configurations is described. Prolog is extended with a number of new predicates to allow the invocation of external tools (*e.g.*, compilers).

As a last example of configuration tool, we cite an expert assistant tool that is the kernel of ENCORES, a knowledge-based programming environment described in [124]: ENCORES has a blackboard architecture that is used to reason on formal documents that are algebraic specifications. Another expert assistant is KDA (Knowledge-based Design Assistant), presented in [169], where the overall system architecture is based on the blackboard model and implemented in POPLOG. KDA can be used for design evaluation and refinement, and is intended to support a software designer in the analysis of alternative designs using multiple evaluation criteria.

More recent complete programming environments that are based on Prolog and offer object oriented abstractions and mechanisms are OBSERV [190] and CAPITL [1]. OBSERV is a prototyping environment able to support and integrate different operational specification formalisms. CAPITL (that stands for Computer-Aided Programming-in-The-Large) is an environment in which the software engineer can use a logic language to describe software objects that are stored in an object database; a planner is then able to search the database to build entities that meet some declarative description.

4.2.2 Maintenance and Reuse

Configuration management and version control are “syntactic” activities, in the sense that they involve module attributes (*e.g.*, date of last modification) independent from their contents. This explains why they can be easily supported by rule-based tools which work on such attributes. Instead, software maintenance and reuse are more difficult problems, because they involve the semantics of modules.

A typical question that should be addressed by a maintenance tool is the following: when a specific library module can be used in a given program? A possible idea consists of associating a logic specification to each available library module, to store these module specifications in a database, and then using a logic language to search the database. However, several questions have to be answered: what is the specification of a module? which queries should be supported for modules’ retrieval? which features should a language offer to support maintenance of large systems decreasing its costs?

These are indeed difficult questions to answer. However, the high declarative level of logic programming has been exploited by several software engineers in prominent industries to develop cost-effective tools.

For instance, a Prolog-based tool supporting software reuse was developed by a Fujitsu team [200]. The tool allowed the retrieval of software modules from a module library. Specification of software modules were formalized using first-order predicate logical formulae. Similarly, in [159] a high-order logic language, Lambda Prolog, was used as query language for searching through program libraries (written in ML) using specification matching. Lambda Prolog was chosen for its higher order theorem proving capabilities useful in specification matching driven by the user.

A more automatic, knowledge-based tool for module retrieval is Code Miner [60], that is a system expert in “scavenging” reusable component from existing software systems. The tool assists the programmer in finding potentially reusable components in programs written in C. The programmer reviews the candidate modules suggested by the tool and judges which ones are truly useful. The source code of modules is represented as facts, whereas all the knowledge (domain-specific, on design, on metric definitions) is codified in Prolog rules.

CAESAR [66] is a system which performs software reuse making inferences on a library of routines. A database contains C modules and their specifications. By dataflow analysis the modules are partitioned in functional slices; inductive logic programming techniques are used to build combinations of functions which frequently occur together in the specifications. These combinations are stored in the library database as well.

A similar project, but for PL/I programs, is described in [31]. In this paper a toolset is described which includes an *extractor* tool, which scans and parses PL/I code, obtaining a relational knowledge base

containing information on program components (*e.g.*, declared variables, procedures, used variables, etc.), and an *abstractor* tool, that is an expert system made of rules codifying the knowledge of a software maintainer which queries the knowledge base built by the extractor.

Even the issue of maintenance of very large software systems has been addressed using a logic programming approach. SOFTM [153] is a software maintenance knowledge-based system that assists a programmer in maintenance and reverse engineering of legacy systems, generating and updating product documentation. In [201] a project of IBM Canada is described, that manages half a million Algol-like lines under the control of 60 people. A Prolog database has been generated, that describes modules and symbols used in the code. Developers can query such a database to improve their comprehension of the system and the quality of the maintenance process. Another industrial project that used a Prolog knowledge base as maintenance database for legacy systems is described in [193]. The problem consisted of managing a large, old software system to adapt and improve it. The knowledge base was built parsing the available documentation concerning a large application already developed. It was then used for implementing special tools like a browser and some functional testers. The authors found that, at least for testing, the use of Prolog-based tools allowed the saving of 20% of overall test efforts.

An important approach to software maintenance is given by reverse engineering tools and methods that aim to improve comprehensibility and maintainability. The REDO project uses a Prolog-based tool to capture the intended semantics of existing COBOL programs, producing functional abstractions and documentation from raw source code [110, 26, 25]. The tool that has been realized relies upon Prolog for expressing both several rewriting strategies and formal verification heuristics. Similar projects, all for reverse engineering of COBOL software, are described in [30, 117].

Finally, in [122] it is shown that the maintenance process should be driven by software complexity metric considerations, that can be developed in a framework specific for logic programming.

4.3 Programming-in-the-many

Most tools examined until now are intended as “intelligent assistant” of a programmer that has to deal with several and/or complex modules. Usually they are intended and are very effective for single users, whereas their scalability as multi-user tools is unclear.

The management and the coordination of several software engineers working on software projects involving several documents is a very complex task that we classify as “programming-in-the-many”.

4.3.1 Project Management

By project management here we do not intend simply resource allocation or budget management, activities for which plenty of tools exist that assist the project manager in the task of controlling the project “out of line” (in fact, a Prolog tool that is an “intelligent assistant” for a project manager is RASP [13], a resource allocator integrated in the PCTE object base).

An ideal tool for managing the software development process, *e.g.*, a project assistant tool able to observe and control the advancement of a project to which several programmers participate, should be able to interact with software written in different programming paradigms and to control the distribution of the different tasks that enact the process.

However, managing and coordinating a set of users and tools concurrently cooperating in a software project is a task beyond the capacity of every sequential language, and *a fortiori* of Prolog. Actually, it is questionable even which kind of languages and systems are more suitable for the task of software process management.

4.3.2 Software Process Modeling

The activity of formally specifying and then executing (“enacting”) the process of building software in which a (large) number of programmers and computing resources is involved, was recently named *software process modeling*. Process modeling is a relatively new research topic in software engineering [41].

The main concern of process modeling is the formalization of a number of customary activities that take place during the production of software objects. An important open problem in software process modeling is the paradigm that should be chosen for the language used to write software process programs. A software process program involves complex and concurrent activities of several agents, that are either programmers or automatic tools. The software process program should specify both the constraint to which all the agents are subject, and the goals that each agent should perform. So a language for software process programming should be both declarative, to specify at a high level what the agents can or cannot do [198], and imperative, to state what the agents should do [149].

A proposal for the use of logic programming in the software process management is found in the SDA project [94], where Prolog was suggested as process programming language [145, 146]. The idea is that the process designer writes a process program specifying in Prolog the rules that govern the software development phases.

Example: A process program can be written as a set of rules that govern the roles of people involved development lifecycle. For instance:

```
% a process program in Prolog [146]
designer_interaction(User_Spec, Arch_design):-
  listup(User_Spec, IO_data_and_Transactions),
  find_threads(IO_data_and_Transactions, Threads),
  write_spec(Threads, Reqs),
  validate(Threads, Reqs, Analysis),
  apply_methods(Reqs, Arch_design),
  propose(User_Spec, Arch_design).
```

This rule states a number of activities that should be executed by a designer to transform a document stating the user requirements into the document describing the architectural design. First IO data and transactions should be listed; then these should be analyzed to find execution threads; the threads should be used to write formal requirements, that should be validated before producing a document specifying the architectural requirements, to be proposed to the user. Note that all activities can backtrack if they fail. □

From our point of view, this use of Prolog for software process modeling is quite naive, especially because it is difficult to integrate and support a process program written in this style inside a real programming environment. In fact, rules like the one above should be seen more as scripts or memos for project management tools, than as an executable specification of development processes. This approach is more evident as it is the main idea at the basis of the design of DesignNet [116], an “intelligent”, knowledge-based tool written in Prolog and able to assist a project leader in coordinating a team of software designers.

Other proposals for the use of Prolog in the activity of software process programming can be found in [77, 33, 85, 196].

Christie [33] suggests that a process model can be designed using a graphic formalism which can then be animated in Prolog to certify formally the development process itself, validating and verifying the development phases that comprise it.

In the EPOS system a software process is written in a process modeling language called SPELL, that is actually a form of Prolog with some object-oriented features [85]. In this case emphasis is on process change and evolution, which are simplified by using rules for describing the software process.

Finally, Welzel [196] describes a process representation technique based on rules that are written in a special graphic form and easily translated in Prolog, so that the process can be simulated and verified. This formalism has been adopted by the SCOPE consortium, which includes a number of European software industries, as a tool for software process evaluation and certification.

When a software process is modeled by Prolog rules there are some important questions that remain without an answer; for instance: how does the process program invoke standard tools? how does it coordinate a team of programmers? how are the complex data structures handled by the software development process modeled?

An approach that tries to give a comprehensive answer to these questions consists of defining the software process as an activity that necessarily takes place within a software development environment. The environment itself is a program: since it specifies the coordination of its users, let us consider the environment specification as the process program. So, building an environment able to offer software process support is like giving an explicit representation of the software process. This is the rationale underlying the notion of process-centered development environments.

For instance, Darwin is an environment for developing systems “ruled by laws” [135]. The main assumption about process programming proposed by Darwin designers is the existence of a law governing the evolution of a system, by constraining the messages allowed to be exchanged among the entities that compose the system. Formally, a law is a restricted Prolog program. Darwin itself is a constraint satisfaction system implemented as a prototype written in Prolog.

A completely different Prolog-based tool is GRAPPLE [84]. It is a plan-based process assistant, that includes primitive environment operators that correspond to atomic actions within the environment. Some primitive actions correspond to tool invocations, while other primitive actions correspond to predefined scripts. These operators may be combined in complex operators to achieve a given goal in a software process phase. Programmers communicate with the environment, which provides both passive (constraints) and active (plans) assistance. Plans are dynamically built by instantiating operators. The environment recognizes user plans and generates system plans. The assistant makes use of explicit knowledge on the software process manipulated by non-monotonic reasoning based on a multivalued logic. GRAPPLE consists of a tool implemented in Prolog, and relies on its inferential capabilities.

The rule-based tools that supports project management suggest that such an approach can be used for designing and implementing the development environments of large-scale software systems. In fact, a definition of the software process is the following [132]:

the software process is a collection of related activities, seen as a coherent process subject to reasoning, involved in the production of a software product.

These activities take place inside a so called *process centered* environment that should actively enact the cooperation of all the agents involved in the software production process. Rule-based coordination of the software process was explored initially in the MARVEL project [90] using an ad hoc notation; then several researchers used notations and environments based on logic programming.

For instance, the approach that sees the process program as a set of coordination rules enacted by a process-centered environment is evident in MERLIN [155]. Such an environment monitors and guides a team of software developers and managers. MERLIN is based on an extension of Prolog that combines backward and forward chaining.

Example:

A knowledge base stores all the information about an software project. For instance, documents to be manipulated are represented by facts of the form `document(Doc_Type,Doc_Name,Status)`, where `Doc_Type` is a document type, and `Doc_Status` is the current status of the document. Access rights to documents are described by facts of the form `work_on(Doc_Type,Doc_Status,R,W,X)`, where `R`, `W`, `X` are lists describing access rights for read, write, and execution operations, respectively.

```
document(module, m1, to_be_edited).
document(specification,m1,specified).
document(error_report, m1, reported).
work_on(module, to_be_edited, [spec, report, review], [module], []).
work_on(review, to_be_reviewed, [spec, module], [review], []).
work_on(object_code, to_be_executed, [spec, module], [], [object_code]).
```

These facts define the document types that a programmer can manipulate. Other facts define the roles of the people involved in the development process, and their responsibilities with respect to the documents. Then, a number of rules define the operating environment inside which every process participant works. For instance, the following rule initializes the working context of a programmer:

```
do_working_context(R,W,X,Item):-
```

Product Specification	[2, 9, 43, 47, 52, 58, 62, 68, 102, 114, 121, 124, 128, 139, 143, 182, 186, 191]
Process Management	[3, 13, 33, 35, 85, 77, 84, 146, 155, 196, 163]
Validation and prototyping	[24, 53, 56, 71, 82, 98, 99, 104, 115, 120, 133, 148, 170, 176, 177, 179, 188, 197]
Design	[8, 4, 38, 44, 63, 106, 109, 118, 147, 160, 162, 202]
Editing and Compiling	[10, 18, 48, 36, 78, 158, 136, 150, 194]
Debugging and testing	[21, 42, 46, 59, 67, 74, 76, 81, 92, 123, 154, 181]
Maintenance and Reuse	[23, 25, 26, 29, 30, 31, 66, 79, 110, 117, 193, 201, 7, 60, 93, 135, 142, 159, 174, 200]

Figure 4: A classification of the projects and tools cited in this paper

```

IF start_working_context
THEN
  CALL(working_context,R,W,X,Doc_Type,Doc_Name, New_Status,Item),
  REMOVE(document(Doc_Type,Doc_Name,_)),
  INSERT(document(Doc_Type, Doc_Name, New_Status)),
  REMOVE(start_working_context);
IF document(module(Object_Name, to_be_compiled))
THEN
  CALL(compiler, Object_Name, Compile_Status),
  REMOVE(document(module(Object_Name, to_be_compiled))),
  INSERT(document(module(Object_Name, Compile_Status))).

```

The first component of this rule specifies that after each change of the document status, the document database is updated accordingly. The second component specifies the automatic invocation of a compiler, again updating the document database with the new correct attributes. \square

Even from this simple example, it is evident that MERLIN is a rule-based software environment that can support a large variety of software processes. These are declaratively defined by set of rules whose combination induces cooperation protocols on a set of programmers.

An issue that is currently not completely clear in the MERLIN project is how a sequential programming language like Prolog can model a software process that is intrinsically distributed and concurrent. On the other hand, this aspect is the main concern of the Oikos project [3, 137]. Oikos is a rule-based software development environment that supports the software process, just like MERLIN. However, the Oikos system language is an extension of the parallel logic language SP [27]. SP itself extends Prolog with a concept of blackboard communication, that allows the explicit specification of parallel coordination of software entities. The use of a parallel language in Oikos is a plus with respect to MERLIN, because it is easy to design and implement distributed coordination protocols. In fact, it has been developed a new distributed programming model based on the concept of multiple tuple spaces, that has been explored as conceptual framework to design process centered software development environments [35].

5 Discussion

This paper includes references to several projects that try to solve very different software engineering problems, and range from almost toy tools built as prototypes in academia to fully-fledged CASE environments built as commercial applications in industry. However, we must try to give a simple classification of such projects, so that readers interested in some specific subjects can find more easily their way.

Figure 4 classifies most of the works we have quoted in this paper according to the lifecycle phase or activity they address. These works show that logic programming has been used in all the phases of the software development lifecycle, aiming at describing and enacting the software process, improving the quality of the specification phase, the incrementality and speed of the design phase, the automation of the testing phase, and the knowledge on the product in the maintenance phase.

Product specification is possibly the lifecycle phase where the use of logic languages is most obvious. It is interesting to note how Prolog has been used as “lingua franca” to animate (implement) requirements specifications written in very different formal languages, like Z or Petri Nets. Parallel logic languages seem also to be very convenient specification formalisms, but they are less widespread than Prolog.

Process modeling is also an obvious application field, because software processes are easily expressed by coordination rules which can manage process-centered environments.

The use of Prolog for validation and prototyping is not surprising: a logic program has both a declarative and a procedural interpretation, and this means that it is easy both to write and to check programs and their properties. Moreover, the availability of powerful Prolog programming environments encourages its use for projects with instable specifications.

The use of logic programming in the design phase has been less studied, probably because there is no notion of modularity in pure logic programming. This drawback is currently being studied and possibly it will be eliminated with the introduction of object-oriented logic languages, which also aim at integrating in the logic paradigm a notion of type. However, the lack of a type system to us does not seem a strong weakness in the design phase, mainly because this means that typeless logic programs are polymorphic and easily reusable.

Logic programming-based editing and compiling is the activity the least represented in our taxonomy: we found a few examples of knowledge-based structure-editors, mainly using constraint logic programming. Possibly the evolution of constraint logic programming will suggest new applications to the coding phase. Instead, the usefulness of logic programming for compiler design and prototyping is widely documented.

Logic programming is quite useful when used for debugging and testing. Prolog debuggers started as very simple tools that progressively evolved to become now sophisticated programming environments. More important, rule-based testing has been successful even for imperative languages.

Finally, the maintenance phase is gaining momentum as a software engineering application field for logic programming. After the development of a number of knowledge-based project databases and configuration tools, the main research trends are now reverse engineering and software process modeling and enactment.

A remarkable common denominator among most of the systems listed in this paper is the instrumental use of logic programming for implementing what a software engineer would call software development environments and tools. An interesting question that naturally arises is whether all these systems could have been realized in some other way. Was the logic language instrumental, beneficial, or even determinant? A naive answer is that whatever you can do in Prolog you can do the same in the another Turing-equivalent programming language. However, this answer misses some important points. A more interesting answer could consist of discussing the relative merits of logic programming with respect to object-oriented programming or functional programming. However, we promised in Sect. 1 that we would not try any comparison among say C++, Miranda, and Prolog from the point of view of a software engineer. Such a comparison would be interesting but not in the scope of our paper. However, we can try to say something about which logic programming features are most promising for a software engineer.

We believe that three basic mechanisms, typical of logic languages, make at least beneficial the use of logic programming technology for designing and building software systems.

At a first level, a software engineer has to shape and handle objects involved in developing software and facts about these objects [104]. First order logic is a powerful data modeling language; it is obtained for free with a logic language. For instance, at this level Prolog can be used as a relational language for defining complex relationships among data structures, and as a language more powerful than a relational query language. Prolog enhances the relational data model, giving the ability to make inferences, by the use of rules, and to prove properties of the whole database, through integrity constraints. Several examples show that programming in-the-small is well supported by a logic database [7]. Such a kind of database is a logic program plus a set of formulas expressing integrity constraints, which define properties to be possessed by the database.

At a second level a software engineer should be able to deal with non-deterministic specifications

and programs. In the configuration phase, for instance, he needs the assistance of planning tools that help him to act on the environment tools [15]. This support level is valuable to programming activities both in-the-small and in-the-large. In the small, Prolog programmers rely on backtracking as a form of implicit invocation of code. Some Prolog environments offer large libraries of reusable code that can be invoked and dynamically loaded by the interpreter. In the large, while single parts of the software development process can be supported by specific tools, global patterns of tool usage should be made explicit for a better exploitation of the environment capabilities. In a rule-based environment typical patterns become environment goals that have to be transformed in production activities by accurate planning. Process knowledge drives the compilation of goals into actions [90].

Again, logic programming technology fits the planning approach to the software development process life cycle. For instance, in Prolog it is easy to build planners, schedulers, and knowledge-based assistants that deal with patterns of software process activities [55, 93, 172]. Deduction and pattern matching mechanisms are embedded into the computational and data model of any logic language, thus easing the associative retrieval of both intensional and extensional information.

Finally, there is a third level related to the coordination of the entities cooperating in a software development environment. It is very important to have powerful communication mechanisms, because such an environment is naturally expressed as a collection of independent cooperating entities that compete for shared resources. It is necessary to coordinate both the programmers and the tools involved in the software production process [149].

The communication mechanisms of parallel logic programming are very helpful in the design of flexible and adaptable environments [65]. Logical unification is the heart of a number of communication techniques, that range from dynamic streams carrying incomplete messages, to partially ordered data channels, and shared dataspace [34]. These mechanisms allow to specify effectively and concisely complex cooperation protocols.

Finally, a last but not least question to answer in this paper is: if logic programming is a so effective and wide-range software technology, why its use in industry is so limited? There are several reasons for this. First and foremost, it is suited mostly for knowledge-based tools, and it is well known that knowledge acquisition is a traditional bottleneck in any specific application. This means that to build really innovative tools is possible, but it is expensive to make them really useful in practice. Another important factor is that academic research concentrated most on theoretic issues, and only recently arose a genuine interest in real-world applications. Also, the interfacing of commercial Prolog environments with the underlying operating system and other programming languages and software tools (*e.g.*, graphic interfaces) improved dramatically in the last five years, and it is still in progress.

6 Conclusions

In this paper we have summarized the most recent trends in logic programming from a software engineering perspective. We have surveyed a number of specific projects in which logic programming is playing a central role.

However, even if these projects show that logic programming is a flexible technology that could and should be used profitably in an industrial framework, we admit that several problems remain open. Some problems are of linguistic nature: logic languages are still evolving incorporating new programming concepts and mechanisms. Other problems are of pragmatic nature: we need more experience in the use of logic programming technology in software engineering activities.

Concerning the use of logic languages as specification languages, we believe that the existing formal support is scarcely practical. We know very few works which develop say a programming logic that supports both reasoning on specifications and their refinement into more efficient programs; a notable exception is [52]. This is strange, because theoretical research on the semantics of logic programming is well developed; however research is more addressed toward static analysis and compilation issues than to refinement of specifications. It should be not difficult to develop a programming logic theory based on this semantical basis, so that refinement of specifications is made possible in practice. Also, theorem provers specialized for formal verification of properties of software systems should be built, possibly

integrated in a fully fledged logic specification and programming environment. This is being done for complex specification notations, like for instance Z, Larch, and PVS. For instance, the Cogito system [16] is a fully fledged integrated environment for Z with a theorem prover, Ergo, and several tools that are written in Prolog. It is unclear why logic languages are not considered as direct targets for similar studies.

From the point of view of language design, there is still room for improvements in the modularity mechanisms offered by current logic programming environments. Probably it will be also necessary to standardize language mechanisms for multiparadigm programming, in order to improve the integration of Prolog systems with both the underlying operating environments and with other languages. For instance, we know of no Prolog implementation which is CORBA-compliant.

Also, a field that is almost completely unexplored is the use of logic languages in the description of the design structure and properties of software architectures.

The most promising course in testing and debugging is also to try to apply to non logic languages the knowledge based debugging models and testing tools already developed for Prolog. For instance, it should be possible to define and implement logic-based debugging query languages for interpreters of imperative languages.

Last but not least, the maintenance phase seems a very promising application field. The development of reverse engineering tools and of maintenance-oriented knowledge bases is just started. Much more experience has to be gained to validate this kind of application.

Acknowledgments. The authors want to gratefully acknowledge all the people that answered their questions and delivered papers through the Internet. This paper would have been impossible to write without the help of the former and the latter.

References

- [1] P. Adams and M. Solomon. An Overview of the CAPITL Software Development Environment. Technical Report TR1143, CS Dept., Univ. of Wisconsin at Madison, 1993.
- [2] V. Ambriola, P. Ciancarini, and A. Corradini. Declarative Specification of the Architecture of a Software Development Environment. *Software: Practice and Experience*, 25(2):143–174, 1995.
- [3] V. Ambriola, P. Ciancarini, and C. Montangero. Enacting software processes in Oikos. In *Proc. ACM SIGSOFT Conf. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 12–23, 1990.
- [4] J. Armstrong, S. Virding, and M. Williams. Use of Prolog for Developing a New Programming Language. In C. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, April 1992. Association for Logic Programming.
- [5] P. Asirelli and P. Inverardi. Enhancing Configuration Facilities in Software Development: A Logic Approach. In *Proc. 1st European Software Eng. Conf. (ESEC)*, volume 289 of *Lecture Notes in Computer Science*, pages 55–63. Springer-Verlag, Berlin, 1987.
- [6] P. Asirelli and P. Inverardi. A logic database to support configuration management in Ada. In S. Tafvelin, editor, *Proc. Int. Conf. Ada-Europe*, pages 19–31, Stockholm, 1987.
- [7] P. Asirelli and P. Inverardi. Using Logic Databases in Software Development Environments. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proc. Int. Workshop on Programming Language Implementation and Logic Programming (PLILP 88)*, volume 348 of *Lecture Notes in Computer Science*, pages 292–293, Orleans, France, 1988. Springer-Verlag, Berlin.
- [8] I. Attali and P. Franchi-Zanettacci. An Inference System Environment for Ada. In S. Tafvelin, editor, *Proc. Int. Conf. Ada-Europe*, pages 3–18, Stockholm, 1987.

- [9] P. Azema et al. Specification and verification of distributed systems using Prolog interpreted Petri Nets. In *Proc. 7th Int. Conf. on Software Engineering*, pages 510–518, Orlando, FL., March 1984.
- [10] R. Ballance, S. Graham, and M. VanDeVanter. The Pan Language-Based Editing System. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.
- [11] B. Beach. Connecting Software Components with Declarative Glue. In *Proc. 14th IEEE Int. Conf. on Software Engineering*, pages 120–137, Melbourne, Australia, 1992.
- [12] G. Bernot, M. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *IEE Software Engineering Journal*, 6, November 1991.
- [13] C. Bertazzoni, M. Gatti, and F. Giannotti. RASP: Resource Allocator for Software Projects. In *Proc. Italian Conf. on Logic Programming*, pages 63–89, Padova, 1990.
- [14] J. Biemann and J. Schultz. Estimating the Number of Test Cases Required to Satisfy the All-du-paths Testing Criterion. In R. Kemmerer, editor, *Proc. 3rd ACM SIGSOFT Symp. on Sw Testing, Analysis, and Verification*, volume 14:8 of *ACM SIGSOFT Software Engineering Notes*, pages 179–186, KeyWest, FL., 1989.
- [15] R. Bisiani, F. Lecouat, and V. Ambriola. A Tool to Coordinate Tools. *IEEE Software*, 5(6):5–15, November 1988.
- [16] A. Bloesch, E. Kazmierczak, P. Kearney, and O. Traynor. Cogito: Methodology and System for Formal Software Development. *Int. Journal on Software Engineering and Knowledge Engineering*, 5(4):599–618, 1995.
- [17] R. Bloomfield and P. Froome. The Application of Formal Methods to the Assessment of High Integrity Software. *IEEE Transactions on Software Engineering*, 12(9):988–993, 1986.
- [18] P. Borrás, D. Clement, T. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. Centaur: The System. In *Proc. 3rd ACM SIGSOFT Symp. on Software Development Environments*, volume 13:5 of *ACM SIGSOFT Software Engineering Notes*, pages 14–24, Boston, 1988.
- [19] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. Meo. Differential Logic Programming. In *Proc. 20th ACM Conf. on Principles of Programming Languages*, pages 359–370, Charleston, NC, 1993.
- [20] L. Bouge, N. Choquet, L. Fribourg, and M. Gaudel. Application of Prolog to Test Sets Generation from Algebraic Specifications. In H. Ehrig, editor, *Formal Methods and Software Development (TAPSOFT 85)*, volume 186 of *Lecture Notes in Computer Science*, pages 261–275, Berlin, March 1985. Springer-Verlag, Berlin.
- [21] L. Bouge, N. Choquet, L. Fribourg, and M. Gaudel. Test Set Generation from Algebraic Specification using Logic Programming. *Journal of Systems and Software*, 6(4):343–360, 1986.
- [22] L. Bouma and H. Walters. Implementing Algebraic Specifications. In J. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, chapter 5, pages 199–282. ACM Press, 1989.
- [23] J. Bowen. From Programs to Object Code and Back Again Using Logic Programming: Compilation and Decompilation. *Software Maintenance - Research and Practice*, 5(4):205–234, 1993.
- [24] J. Bowen, H. Jifeng, and P. Pandya. An Approach to Verifiable Compiling Specification and Prototyping. In P. Deransart and J. Maluszynski, editors, *Proc. Int. Symposium on Programming Language Implementation and Logic Programming (PLILP 90)*, volume 456 of *Lecture Notes in Computer Science*, pages 45–59, Sweden, 1990. Springer-Verlag, Berlin.

- [25] P. Breuer. The Art of Computer un-Programming: Reverse Engineering in Prolog. In G. Comyn, N. Fuchs, and M. Ratcliffe, editors, *Logic Programming in Action*, volume 636 of *Lecture Notes in Computer Science*, pages 290–302. Springer-Verlag, Berlin, September 1992.
- [26] P. Breuer and K. Lano. Creating Specifications from Code: Reverse Engineering Techniques. *Software Maintenance - Research and Practice*, 3:145–162, 1991.
- [27] A. Brogi and P. Ciancarini. The concurrent language Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.
- [28] F. Brooks. No Silver Bullet: Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4):10–19, April 1987.
- [29] J. Callahan and J. Purtilo. A Packaging System for Heterogeneous Execution Environments. *IEEE Transactions on Software Engineering*, 17(6):626–635, June 1991.
- [30] G. Canfora, A. Cimitile, and U. deCarlini. A Logic-Based Approach to Reverse Engineering Tools Production. *IEEE Transactions on Software Engineering*, 18(12):1053–1064, December 1992.
- [31] G. Canfora, A. Cimitile, and M. Tortorella. Prolog for Software Maintenance. In *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 478–486, Rockville, Maryland, 1995. Knowledge Systems Institute.
- [32] V. Carchiolo, A. Faro, and M. Malgeri. A Tool for the Performance Analysis of Concurrent Systems. In C. Ratray, editor, *Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 121–139. Springer-Verlag, Berlin, 1989.
- [33] A. Christie. *Software Process Automation*. Springer-Verlag, Berlin, 1995.
- [34] P. Ciancarini. Parallel Programming with Logic Languages: a Survey. *Computer Languages*, 17(4):213–240, 1992.
- [35] P. Ciancarini. Coordinating Rule-Based Software Processes with ESP. *ACM Transactions on Software Engineering and Methodology*, 2(3):203–227, 1993.
- [36] L. Cohen and T. Hickey. Parsing and Compiling Using Prolog. *ACM Transactions on Programming Languages and Systems*, 9(3):125–163, April 1987.
- [37] A. Colmerauer. Les grammaires des metamorphose. Technical report, Groupe d’Intelligence Artificielle, Univ. of Marseille-Luminy, 1975.
- [38] M. Consens, A. Mendelzon, and A. Ryman. Visualizing and Querying Software Structures. In *Proc. 14th IEEE Int. Conf. on Software Engineering*, pages 138–156, Melbourne, Australia, 1992.
- [39] P. Coscia, P. Franceschi, G. Levi, G. Sardu, and L. Torre. Inference Engine Definition and Compilation in the Epsilon Logic Programming Environment. In R. Kowalski and K. Bowen, editors, *Proc. 5th Int. Conf. on Logic Programming*, pages 359–373, Seattle, WA, 1988. MIT Press, Cambridge, MA.
- [40] M. Costa, R. Cunningham, and J. Booth. Logical Animation. In *Proc. 12th IEEE Int. Conf. on Software Engineering (ICSE)*, pages 144–149, 1990.
- [41] B. Curtis, M. Kellner, and J. Over. Process Modeling. *Communications of the ACM*, 35(9):75–90, September 1992.
- [42] P. Dauchy and B. Marre. Test Data Selection from Algebraic Specifications: Application to an Automatic Subway Module. In *Proc. 3rd European Software Eng. Conf. (ESEC 91)*, volume 550 of *Lecture Notes in Computer Science*, pages 80–100, Milan, Italy, 1991. Springer-Verlag, Berlin.

- [43] R. Davis. Runnable Specifications as Design Tool. In K. Clark and S. Tarnlund, editors, *Logic Programming*, pages 141–149. Academic Press, 1982.
- [44] A. Davison. A Survey of Logic Programming-based Object Oriented Languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Trends in Object-Based Concurrent Computing*, pages 42–106. MIT Press, Cambridge, MA, 1993.
- [45] T. Dean and J. Condy. A Syntactic Theory of Software Architecture. *IEEE Transactions on Software Engineering*, 21(4):302–313, April 1995.
- [46] W. Deason, D. Brown, K. Chang, and J. Cross. A Rule-Based Software Test Data Generator. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):108–117, March 1991.
- [47] M. Degl’Innocenti, G. Ferrari, G. Pacini, and F. Turini. RSF: A Formalism for Executable Requirement Specification. *IEEE Transactions on Software Engineering*, 16(11):1235–1246, November 1990.
- [48] A. DeLucia, A. Imperatore, M. Napoli, G. Tortora, and M. Tucci. The Software Development Workbench WSDW. In *Proc. 4th IEEE Conf. on Software Engineering and Knowledge Engineering (SEKE)*, pages 213–221. IEEE Computer Society Press, 1992.
- [49] R. Denney. Test-case generation from Prolog-based specifications. *IEEE Software*, 8(2):49–57, March 1991.
- [50] P. Deransart and G. Ferrand. An Operational Formal Definition of Prolog: A Specification Method and Its Application. *New Generation Computing*, 10(2):121–172, 1992.
- [51] P. Deransart and J. Maluszynski. A Grammatical View of Logic Programming. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proc. Int. Workshop on Programming Language Implementation and Logic Programming (PLILP 88)*, volume 348 of *Lecture Notes in Computer Science*, pages 219–249, Orleans, France, 1988. Springer-Verlag, Berlin.
- [52] Y. Deville. *Logic Programming: Systematic Program Development*. Addison-Wesley, 1990.
- [53] A. Dick, P. Krause, and J. Cozens. Computer aided transformation of Z into Prolog. In J. Nicholls, editor, *Proc. 4th Z Users Workshop*, Workshops in Computing, pages 71–85, Oxford, 1989. Springer-Verlag, Berlin.
- [54] A. Diller. *Z: An Introduction to Formal Methods*. Wiley, New York, 1990.
- [55] M. Dinçbas, P. VanHentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 693–702, Tokyo, Japan, 1988.
- [56] T. Docker. SAME: A Structured Analysis tool and its implementation in Prolog. In R. Kowalski and K. Bowen, editors, *Proc. 5th Int. Conf. and Symp. on Logic Programming*, pages 82–95. MIT Press, Cambridge, MA, 1988.
- [57] V. Doma and R. Nicholl. EZ: A System for Automatic Prototyping of Z Specifications. In S. Prehn and W. Toetenel, editors, *VDM 91: Formal Software Development Methods*, volume 551 of *Lecture Notes in Computer Science*, pages 189–203, Noordwijkerhout, October 1991. Springer-Verlag, Berlin.
- [58] Y. Dotan and B. Arazi. Using Flat Concurrent Prolog in System Modeling. *IEEE Transactions on Software Engineering*, 17(6):493–513, June 1991.

- [59] M. Ducassé. A general trace query mechanism based on Prolog. In M. Bruynooghe and M. Wirsing, editors, *Proc. Int. Symposium on Programming Language Implementation and Logic Programming (PLILP 92)*, volume 631 of *Lecture Notes in Computer Science*, pages 400–414, Leuven, Belgium, 1992. Springer-Verlag, Berlin.
- [60] M. Dunn and J. Knight. Automating the Detection of Reusable Parts in Existing Software. In *Proc. 15th Int. Conf. on Software Engineering*, pages 381–390, Baltimore, Md, May 1993.
- [61] A. EdDbali and P. Deransart. Software Formal Specification by Logic Programming: The Example of Standard Prolog. In G. Comyn, N. Fuchs, and M. Ratcliffe, editors, *Logic Programming in Action*, volume 636 of *Lecture Notes in Computer Science*, pages 278–289. Springer-Verlag, Berlin, September 1992.
- [62] F. Ferrucci, G. Nota, G. Pacini, S. Orefice, and G. Tortora. On Refinement of Logic Specifications. *Int. Journal on Software Engineering and Knowledge Engineering*, 2(3):433–448, 1992.
- [63] I. Foster. A Declarative State-Transition System. *Journal of Logic Programming*, 10:45–67, 1989.
- [64] I. Foster. Implementation of a Declarative State-Transition System. *Software: Practice and Experience*, 19(4):351–370, 1989.
- [65] I. Foster. *System Programming in Parallel Logic Languages*. Prentice-Hall, 1990.
- [66] G. Fouque and S. Matwin. A Case-Based Approach to Software Reuse. *Journal of Intelligent Information Systems*, 2(2):88–121, June 1993.
- [67] P. Fung. Applying Formal Semantics to User Modeling. *Journal of Artificial Intelligence in Education*, 3(3):315–345, 1992.
- [68] F. Garzotto, C. Ghezzi, D. Mandrioli, and A. Morzenti. On the specification of real-time systems using logic programming. In *Proc. 1st European Software Eng. Conf. (ESEC)*, volume 289 of *Lecture Notes in Computer Science*, pages 180–190. Springer-Verlag, Berlin, 1987.
- [69] D. Gilbert. Specification and Implementation of Concurrent Systems using PARLOG. In C. Rattray, editor, *Specification and Verification of Concurrent Systems*, Workshops in Computing, pages 455–474. Springer-Verlag, Berlin, 1989.
- [70] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:139–185, 1991.
- [71] A. Giovini, F. Morando, and A. Capani. Implementation of a Toolset for Prototyping Algebraic Specifications of Concurrent Systems. In H. Kirchner and G. Levi, editors, *Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 335–349, Volterra, Italy, September 1992. Springer-Verlag, Berlin.
- [72] T. Goble. *Structured Systems Analysis through Prolog*. Prentice-Hall, 1989.
- [73] J. Goguen and J. Meseguer. Eqlog: Equality, Types and Generic Modules for Logic Programming. In D. de Groot and G. Lindstrom, editors, *Logic Programming, Functions, Relations and Equations*, pages 295–363. Prentice-Hall, 1986.
- [74] M. Gorlick, C. Kesselman, D. Marotta, and D. Parker. Mockingbird: A Logical Methodology for Testing. *Journal of Logic Programming*, 8(1-2):95–119, 1990.
- [75] N. Habra. From Abstract Data Types to Logic Programs: a Transformational Approach. In J. Jacquet, editor, *Constructing Logic Programs*, pages 251–278. Wiley, New York, 1993.
- [76] D. Hamlet. Implementing Prototype Testing Tools. *Software: Practice and Experience*, 25(4):347–371, April 1995.

- [77] D. Heimbigner. P4: A Logic Language for Process Programming. In *Proc. 5th Int. Software Process Workshop*, Kennebunkport, Maine, 1989.
- [78] P. Henriques. A Semantic Evaluator Generating System in Prolog. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proc. Int. Workshop on Programming Language Implementation and Logic Programming (PLILP 88)*, volume 348 of *Lecture Notes in Computer Science*, pages 201–217, Orleans, France, 1988. Springer-Verlag, Berlin.
- [79] C. Ho, H. Manabe, and H. Yabe. Test Consulting System for Switching Systems. In *Proc. 15th Int. Conf. COMPSAC '91*, pages 130–135. IEEE Computer Society Press, 1991.
- [80] D. Hoffman and C. Brealey. Module Test Case Generation. In R. Kemmerer, editor, *Proc. 3rd ACM SIGSOFT Symp. on Sw Testing, Analysis, and Verification*, volume 14:8 of *ACM SIGSOFT Software Engineering Notes*, pages 97–102, KeyWest, Fl., 1989.
- [81] D. Hoffman and P. Strooper. Automated Module Testing in Prolog. *IEEE Transactions on Software Engineering*, 17(9):934–943, September 1991.
- [82] S. Honiden, N. Uchihira, and T. Kasuya. Software Prototyping with MENDEL. In E. Wada, editor, *Logic Programming '85*, volume 221 of *Lecture Notes in Computer Science*, pages 108–116, Tokyo, Japan, 1985. Springer-Verlag, Berlin.
- [83] J. Hsiang and M. Srivas. A Prolog Environment for Developing and Reasoning about Data Types. In H. Ehrig et al., editors, *Formal Methods and Software Development (TAPSOFT 85)*, volume 186 of *Lecture Notes in Computer Science*, pages 276–293, Berlin, March 1985. Springer-Verlag, Berlin.
- [84] K. Huff and V. Lesser. A Plan-Based Intelligent Assistant that Supports the Software Development Process. In *Proc. 3rd ACM SIGSOFT Symp. on Software Development Environments*, volume 13:5 of *ACM SIGSOFT Software Engineering Notes*, pages 97–106, 1988.
- [85] M. Jaccheri and R. Conradi. Techniques for Process Model Evolution in EPOS. *IEEE Transactions on Software Engineering*, 19(12):1145–1156, December 1993.
- [86] J. Jaffar and M. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19-20:503–581, 1994.
- [87] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–394, July 1992.
- [88] S. Jarzabeck. PQL: a language for specifying abstract program views. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC)*, volume 989 of *Lecture Notes in Computer Science*, pages 324–341, Sitges, Spain, September 1995. Springer-Verlag, Berlin.
- [89] K. Kahn. Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs. Technical Report SSL91-16/P91-00143, XEROX PARC, Palo Alto, CA, 1991.
- [90] G. Kaiser, P. Feiler, and S. Popovich. Intelligent Assistance for Software Development and Maintenance. *IEEE Software*, 5(5):40–49, May 1988.
- [91] D. Katzenellenbogen, S. Cohen, and E. Shapiro. Architecture of a Distributed Window System and its FCP Implementation. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 101–139. MIT Press, Cambridge, MA, 1987.
- [92] S. Khanna. Logic Programming for Software Verification and Testing. *The Computer Journal*, 34(4):350–357, 1991.

- [93] T. Kielmann. Using Prolog for Software System Maintenance. In K. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, April 1992. Association for Logic Programming.
- [94] K. Kishida et al. SDA: A Novel Approach to Software Environment Design and Construction. In *Proc. 10th Int. Conf. on Software Engineering*, pages 69–79, Singapore, April 1988.
- [95] R. Knott. Using Prolog to animate mathematics. In D. Brough, editor, *Logic Programming: New Frontiers*, Intellect Books, chapter 8, pages 173–188. Kluwer Academic Publishers, Dordrecht/Boston/London, 1992.
- [96] R. Knott and P. Krause. An Approach to Animating Z Using Prolog. Technical Report Report A1.1, Alvey Project SE/065, University of Surrey, 1988.
- [97] R. Knott and P. Krause. The implementation of Z specifications using program transformation systems: The SuZan project. In C. Rattray and R. Clark, editors, *The Unified Computation Laboratory*, volume 35 of *IMA Conference Series*, pages 207–220, Oxford, UK, 1992. Clarendon Press.
- [98] H. Komorowski and J. Maluszynski. Logic Programming and Rapid Prototyping. *Science of Computer Programming*, 9:179–205, 1987.
- [99] D. Kourie. The Design and Use of a Prolog Trace Generator for CSP. *Software: Practice and Experience*, 17(7):423–438, July 1987.
- [100] R. Kowalski. Predicate Logic as a Programming Language. In *Proc. IFIP Conference*, pages 556–574. North-Holland, 1974.
- [101] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22:424–436, 1979.
- [102] R. Kowalski. The relation between logic programming and logic specification. In C. Hoare and J. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 1–24. Prentice-Hall, 1985.
- [103] R. Kowalski. Software engineering and artificial intelligence in new generation computing. *Future Generation Computer Systems*, 1(1):39–49, 1985.
- [104] C. Kung. Conceptual Modelling in the Context of Software Development. *IEEE Transactions on Software Engineering*, 15(10):1176–1187, October 1989.
- [105] T. Kurozumi. Overview of the Ten Years of the FGCS Project. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pages 9–19, Tokyo, Japan, 1992.
- [106] P. Kursawe. How to Invent a Prolog Machine. *New Generation Computing*, 5(1):97–114, 1987.
- [107] A. Kusalik. Specification and Initialization of a Logic Computer System. *New Generation Computing*, 4(2):189–209, 1986.
- [108] A. Kusalik. Secondary Storage in a Concurrent Logic Programming Environment. *Journal of Systems and Software*, 11:31–44, 1990.
- [109] A. Lakhota. Rule-based Approach to Computing Module Cohesion. In *Proc. 15th IEEE Conf. on Software Engineering*, pages 35–44. IEEE Computer Society Press, 1993.
- [110] K. Lano and P. Breuer. Using Prolog for Reverse-Engineering and Validation. In L. Sterling and U. Yalcinalp, editors, *Proc. Workshop in Expert Systems, AI and Sw Eng. Applications*, ILPS, San Diego, Ca, October 1991. Ass. for Logic Programming.

- [111] G. Lazarev. *Why Prolog? Justifying Logic Programming for Practical Applications*. Prentice-Hall, 1989.
- [112] G. Lazarev and W. Gresow. Logic programming as a Software Engineering Tool. In *Proc. CO-COMO/WICOMO Users' Group Meeting*, pages 290–302, Wang Institute, Tyngsboro, MA, May 1985.
- [113] P. Lee, F. Pfenning, G. Rollins, and W. Scherlis. The Ergo Support System: an Integrated Set of Tools for Prototyping Integrated Environments. In *Proc. ACM SIGSOFT 3rd Symp. on Software Development Environments*, volume 13:5 of *ACM SIGSOFT Software Engineering Notes*, pages 25–34, 1988.
- [114] S. Lee and S. Sluizer. An Executable Language for Modeling Simple Behavior. *IEEE Transactions on Software Engineering*, 17(6):527–543, June 1991.
- [115] U. Leibrandt and P. Schnupp. An Evaluation of Prolog as a Prototyping System. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Zullighoven, editors, *Approaches to Prototyping*, pages 424–433. Springer-Verlag, Berlin, 1984.
- [116] L. Liu and E. Horowitz. A Formal Model for Software Project Management. *IEEE Transactions on Software Engineering*, 15(10):1280–1293, 1989.
- [117] Z. Liu. Automating Software Evolution. *Int. Journal on Software Engineering and Knowledge Engineering*, 5(1):73–88, March 1995.
- [118] C. Lucena and P. Alencar. A Formal Description of Evolving Software Systems Architectures. *Science of Computer Programming*, 24(1):41–62, February 1995.
- [119] D. Luckham and F. vonHenke. An Overview of Anna, a Specification language for Ada. *IEEE Software*, 2(2):9–22, March 1985.
- [120] Luqi and E. Cooke. How to Combine Nonmonotonic Logic and Rapid Prototyping to Help Maintain Software. *Int. Journal on Software Engineering and Knowledge Engineering*, 5(1):89–118, March 1995.
- [121] B. Macias and S. Pulman. A method for controlling the production of specifications in natural language. *The Computer Journal*, 38(4):310–318, 1995.
- [122] Z. Markusz and A. Kaposi. Complexity control in logic based programming. *The Computer Journal*, 28(5):487–95, 1985.
- [123] B. Marre. Toward Automatic Test Data Set Selection using Algebraic Specifications and Logic Programming. In K. Furukawa, editor, *Proc. 8th Int. Conf on Logic Programming*, pages 202–219, Paris, 1991. MIT Press, Cambridge, MA.
- [124] D. McArthur. ENCORES: an environment for constructing or reasoning with engineered software. In P. Brereton, editor, *Software Engineering Environments*, pages 69–78. Ellis Horwood, Chichester, 1988.
- [125] T. Mccluskey, M. Porteus, Y. Naik, C. Taylor, and S. Jones. A Requirements Capture Method and its use in an Air Traffic Control System. *Software: Practice and Experience*, 25(1):47–71, January 1995.
- [126] C. Meadows. Applying formal methods to the analysis of a key management protocol. *Journal of Computer Security*, 1(1):5–36, 1992.
- [127] M. Meier and J. Schimpf. An Architecture for Prolog Extensions. In E. Lamma and P. Mello, editors, *Proc. 3rd Workshop on Extensions of Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 319–338. Springer-Verlag, Berlin, 1992.

- [128] L. Mekly, M. Todd, and M. Yuhas. Automated Test Case Generation for Requirements Knowledge Base. In *Proc. ATT/NCR Performance analysis and Sw Tools (TIES)*, pages 1–36, Naperville, IL, 1992.
- [129] L. Mekly and M. Yuhas. A Logic Programming Approach to Requirements Modeling and Automated Test Generation (Poster). In *Proc. 10th Int. Conf. on Logic Programming*, page 849, Budapest, Hungary, 1993. MIT Press, Cambridge, MA.
- [130] P. Mello and A. Natali. Extending Prolog with Modularity, Concurrency and Metarules. *New Generation Computing*, 10(4):335–360, 1992.
- [131] P. Mello, A. Natali, and C. Ruggieri. Logic programming in a software engineering perspective. In E. Lusk and R. Overbeek, editors, *Proc. North American Conf. on Logic Programming*, pages 441–458. MIT Press, Cambridge, MA, 1989.
- [132] P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulating software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, 2(3):283–294, September 1990.
- [133] J. Millen, S. Clark, and S. Freedman. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, 13(2):274–288, February 1987.
- [134] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(2):79–108, 1989.
- [135] N. Minsky and D. Rozenstein. Configuration Management by Consensus: An Application of Law-Governed Systems. In R. Taylor, editor, *Proc. 4th ACM SIGSOFT Symp. on Software Development Environments*, volume 15:6 of *ACM SIGSOFT Software Engineering Notes*, pages 44–55, December 1990.
- [136] J. Monin. A compiler written in Prolog: the Veda experience. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proc. Int. Workshop on Programming Language Implementation and Logic Programming (PLILP 88)*, volume 348 of *Lecture Notes in Computer Science*, pages 119–131, Orleans, France, 1988. Springer-Verlag, Berlin.
- [137] C. Montangero and F. Scarselli. Software Process Monitoring Mechanisms in Oikos. *Int. Journal on Software Engineering and Knowledge Engineering*, 4(4):481–499, 1994.
- [138] J. Moreno and M. Rodriguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.
- [139] A. Morzenti, D. Mandrioli, and C. Ghezzi. TRIO, a logic language for executable specifications of real time systems. *Journal of Systems and Software*, 12(2):107–123, 1990.
- [140] C. Moss. *Prolog++*. *The Power of ObjectOriented and Logic Programming*. Addison-Wesley, 1994.
- [141] C. Moss and K. Bowen, editors. *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, April 1992. Association for Logic Programming.
- [142] D. Nardi and M. Tucci. Building Tools for Software Engineering with AI Techniques. *Applied Artificial Intelligence*, 4:283–307, 1990.
- [143] C. Niskier, T. Maibaum, and D. Schwabe. A Look through PRISMA: Towards Pluralistic Knowledge-based Environments for Software Specification Acquisition. In C. Potts, editor, *Proc. 5th IEEE Int. Workshop on Software Specification and Design*, pages 128–136, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

- [144] C. Niskier, T. Maibaum, and D. Schwabe. A Pluralistic Knowledge-Based Approach to Software Specification. In *Proc. 2nd European Software Engineering Conference (ESEC)*, volume 387 of *Lecture Notes in Computer Science*, pages 411–423, Coventry, UK, 1989. Springer-Verlag, Berlin.
- [145] K. Ochimizu and A. Ohki. A Prolog-Based Approach to SDA Prototyping. In *Proc. 21st Hawaii Conf. on System Sciences*, 1988.
- [146] A. Ohki and K. Ochimizu. Process Programming with Prolog. In *Proc. ACM Software Process Workshop*, volume 14:4 of *ACM SIGSOFT Software Engineering Notes*, pages 118–121, 1989.
- [147] A. Okkonen, A. Auer, M. Levanto, J. Okkonen, and J. Kalaoja. Sokrates-SA: A Formal Method for Specifying Real-Time Systems. *Microprocessing and Microprogramming*, 27:1–5, 1989.
- [148] I. O’Neill. Industrial report on Applications of Prolog in Software Validation and Verification Tools. In *Proc. Int. Conf. on Practical Applications of Prolog*, pages 479–488, Paris, France, 1995.
- [149] L. Osterweil. Software Processes are Software Too. In *Proc. 9th IEEE Int. Conf. on Software Engineering*, pages 2–13, 1987.
- [150] J. Paakki. PROFIT: A System integrating Logic Programming and Attribute Grammars. In *3rd Int. Symp. on Programming Language Implementation and Logic Programming (PLILP 91)*, volume 528 of *Lecture Notes in Computer Science*, pages 243–254, Passau, Germany, 1991. Springer-Verlag, Berlin.
- [151] J. Paakki. Prolog in Practical Compiler Writing. *The Computer Journal*, 34(1):64–72, 1991.
- [152] J. Paakki. Attribute Grammar Formalisms - A High Level Methodology in Language Implementation. *ACM Computer Surveys*, 27(2):196–256, June 1995.
- [153] L. Pau and J. Kristinsson. SOFTM: A Software Maintenance Expert System in Prolog. *Software Maintenance - Research and Practice*, 2:87–111, 1990.
- [154] H. Pesch, P. Schnupp, H. Schaller, and A. Spirik. Test Case Generation using Prolog. In *Proc. 8th IEEE Int. Conf. on Software Engineering*, pages 252–258, London, 1985. IEEE Computer Society Press.
- [155] B. Peuschel, W. Schaefer, and S. Wolf. A Knowledge-Based Software Development Environment Supporting Cooperative Work. *Int. Journal on Software Engineering and Knowledge Engineering*, 2(1):79–106, 1992.
- [156] T. Pressburger and D. Smith. Knowledge-based software development tools. In P. Brereton, editor, *Software Engineering Environments*, pages 79–103. Ellis Horwood, Chichester, 1988.
- [157] U. Reddy. Transformational Derivation of Programs Using the Focus System. In *Proc. 3rd ACM Symp. on Software Development Environments*, volume 13:5 of *ACM SIGSOFT Software Engineering Notes*, pages 163–172, 1988.
- [158] G. Riedewald and U. Lammel. Using an Attribute Grammar as a Logic Program. In P. Deransart, B. Lorho, and J. Maluszynski, editors, *Proc. Int. Workshop on Programming Language Implementation and Logic Programming (PLILP 88)*, volume 348 of *Lecture Notes in Computer Science*, pages 161–179, Orleans, France, 1988. Springer-Verlag, Berlin.
- [159] E. Rollins and J. Wing. Specifications as Search Keys for Software Libraries. In K. Furukawa, editor, *Proc. 8th Int. Conf. on Logic Programming*, pages 173–187, Paris, 1991. MIT Press, Cambridge, MA.
- [160] M. Rueher. From Specification to Design: An Approach based on Rapid Prototyping. In *Proc. 4th IEEE Int. Workshop on Software Specification and Design*, pages 126–133. IEEE Computer Society Press, 1987.

- [161] M. Rueher and B. Legeard. Which Role for CLP in Software Engineering? An Investigation on the Basis of first Applications. In C. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, April 1992. Association for Logic Programming.
- [162] S. Safra and E. Shapiro. Metainterpreters for Real. In H.-J. Kugler, editor, *Proc. IFIP Conference*, pages 271–278. North-Holland, 1986.
- [163] W. Schafer and S. Wolf. Cooperation Patterns for process-centered Software Development Environments. In *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 454–462, Rockville, Maryland, 1995. Knowledge Systems Institute.
- [164] Z. Scherz, O. Maler, and E. Shapiro. Learning with Prolog: a new approach. In J. Briggs and J. Dean, editors, *Prolog, Children and Students*, Fifth Generation Computing in Education, pages 91–103. Kogan Page, 1988.
- [165] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, MA, 1983.
- [166] E. Shapiro. The Family of Concurrent Logic Languages. *ACM Computer Surveys*, 21(3):412–510, September 1989.
- [167] E. Shapiro and A. Takeuchi. Object-Oriented Programming in Concurrent Prolog. *New Generation Computing*, 1(1):25–49, 1983.
- [168] E. Shapiro and D. Warren. The Fifth Generation Project: Personal Perspectives (special issue). *Communications of the ACM*, 36(3):46–100, March 1993.
- [169] H. Sharp. KDA - A Tool for Automatic Design Evaluation and Refinement Using the Blackboard Model of Control. In *Proc. 10th Int. Conf. on Software Engineering*, pages 407–416, Singapore, April 1988.
- [170] D. Sidhu and C. Crall. Executable Logic Specifications for Protocol Service Interfaces. *IEEE Transactions on Software Engineering*, 14(1):98–121, January 1988.
- [171] W. Silverman, M. Hirsch, A. Hourii, and E. Shapiro. The Logix System Manual. In E. Shapiro, editor, *Concurrent Prolog: Collected Papers*, pages 46–77. MIT Press, Cambridge, MA, 1987.
- [172] P. Singleton and P. Brereton. A Logic Database for Software Release Engineering. In *Proc. IEEE Conf. on Software Maintenance*, pages 206–213. IEEE Computer Society Press, 1990.
- [173] P. Singleton and P. Brereton. Building Software by Deduction: Why and How. Technical Report TR92-17, Keele University Computer Science Dept., UK, 1992.
- [174] P. Singleton and P. Brereton. A Case for Declarative Programming-in-the-Large. In *Proc. 5th IEEE Conf. on Software Engineering and Knowledge Engineering (SEKE)*, California, 1993.
- [175] J. Spivey. *The Z Notation. A Reference Manual*. Prentice-Hall, 2 edition, 1992.
- [176] K. Steer. Testing Data Flow Diagrams with PARLOG. In R. Kowalski and K. Bowen, editors, *Proc. 5th Int. Conf. and Symp on Logic Programming*, pages 96–110. MIT Press, Cambridge, MA, 1988.
- [177] S. Stepney and S. Lord. Formal Specification of an Access Control System. *Software: Practice and Experience*, 17(9):575–593, September 1987.
- [178] S. Stepney, D. Whitley, D. Cooper, and C. Grant. A Demonstrably Correct Compiler. *Formal Aspects of Computing*, 3(1):58–101, 1991.
- [179] L. Sterling, P. Ciancarini, and T. Turnidge. On the Animation of Not Executable Specifications by Prolog. *Int. Journal on Software Engineering and Knowledge Engineering*, 6(1):(to appear), 1996.

- [180] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.
- [181] P. Strooper and D. Hoffman. Prolog Testing of C Modules. In V. Saraswat and K. Ueda, editors, *Proc. Int. Symposium on Logic Programming*, pages 596–608, SanDiego, USA, 1991. MIT Press, Cambridge, MA.
- [182] N. Suzuki. Experience with Specification and Verification of a Complex Computer Using Concurrent Prolog. In M. vanCaneghem and D. Warren, editors, *Logic Programming and Its Applications*, pages 105–116. Ablex, 1986.
- [183] A. Takeuchi and K. Furukawa. Partial evaluation of Prolog programs and its application to metaprogramming. In H. Kugler, editor, *Proc. IFIP 86*, pages 415–420. North-Holland, 1986.
- [184] R. Terwilliger and R. Campbell. PLEASE: a Language for Incremental Software Development. In *Proc. 4th IEEE Int. Workshop on Software Specification and Design*, pages 249–256. IEEE Computer Society Press, 1987.
- [185] R. Terwilliger and R. Campbell. An Early Report on Encompass. In *Proc. 10th Int. Conf. on Software Engineering*, pages 344–354, Singapore, April 1988.
- [186] R. Terwilliger and R. Campbell. PLEASE: Executable Specifications for incremental software development. *Journal of Systems and Software*, 10:97–112, 1989.
- [187] R. Trehan and P. Wilk. A parallel chart parser for the committed choice non deterministic logic languages. In R. Kowalski and K. Bowen, editors, *Proc. 5th Int. Conf. and Symp on Logic Programming*, pages 212–232. MIT Press, Cambridge, MA, 1988.
- [188] J. Tsai, T. Weigert, and H. Jang. A Hybrid Knowledge Representation as a Basis of Requirements Specification and Specification Analysis. *IEEE Transactions on Software Engineering*, 18(12):1076–1100, December 1992.
- [189] T. Tse, T. Chen, and C. Kwok. The use of prolog in the modelling and evaluation of structure charts. *Information and Software Technology*, 36(1):23–33, 1994.
- [190] S. Tyszberowicz and A. Yehudai. OBSERV - A Prototyping Language and Environment. *ACM Transactions on Software Engineering and Methodology*, 1(3):269–309, July 1992.
- [191] H. Ural. Specifications of Distributed Systems in Prolog. *Journal of Systems and Software*, 11:143–154, 1990.
- [192] P. VanHentenryck, H. Simonis, and M. Dincbas. Constraint Satisfaction Using Logic Programming. *Artificial Intelligence*, 58(1):113–159, 1992.
- [193] J. Vaucher, G. Bochmann, B. Lefebvre, K. Lee, S. Vella, and M. Wu. Prolog for Industrial Software Development. In C. Moss and K. Bowen, editors, *Proc. 1st Conf. on The Practical Application of Prolog*, London, England, April 1992. Association for Logic Programming.
- [194] D. Warren. Logic Programming and Compiler Writing. *Software: Practice and Experience*, 10(2):97–125, February 1980.
- [195] P. Wegner. Dimensions of Object-Oriented Modeling. *IEEE Computer*, 25(10):22–39, October 1992.
- [196] D. Welzel. A rule-based process representation technique for software process evaluation. *Information and Software Technology*, 35(10):603–610, 1993.
- [197] M. West and B. Eaglestone. Software Development: Two Approaches to Animation of Z specifications using Prolog. *IEE Software Engineering Journal*, 7(4):264–276, July 1992.

- [198] L. Williams. Software Process Modeling: a Behavioral Approach. In *Proc. 10th Int. Conf. on Software Engineering*, pages 174–186, Singapore, April 1988.
- [199] J. Wing. A Specifier’s Introduction to Formal Methods. *IEEE Computer*, 23(9):8–24, September 1990.
- [200] H. Yoshida, H. Kato, and M. Sugimoto. Retrieval of Software Module Specifications Using First Order Predicate Logical Formulae. In E. Wada, editor, *Logic Programming ’85*, volume 221 of *Lecture Notes in Computer Science*, pages 117–127, Tokyo, Japan, 1985. Springer-Verlag, Berlin.
- [201] B. Yu. LARGE Software System Maintenance. In *6th Annual Knowledge Based Software Engineering Conference*, Syracuse, New York, September 1991. IEEE Computer Society Press.
- [202] C. Zaniolo. Object-Oriented Programming in Prolog. In *Proc. IEEE Symp. on Logic Programming*, pages 265–270, Atlantic City, NJ, 1984. IEEE Computer Society Press.
- [203] P. Zave. A Compositional Approach to Multiparadigm Programming. *IEEE Software*, 6(5):15–27, September 1989.