1997

Logic programming and software maintenance

Cooke, Daniel; Luqi

J.C. Balzer AG, Science Publishers

D. Cooke, Luqi, "Logic programming and software maintenance," Annals of Mathematics and Artificial Intelligence, v.21 (1997), pp. 221-229
http://hdl.handle.net/10945/52547

Annals of Mathematics and Artificial Intelligence 21 (1997) 221–229

Logic programming and software maintenance *

Daniel Cooke [a] and Luqi [b]

[a] *Computer Science, University of Texas at El Paso, El Paso, TX 79968, USA*
[b] *Computer Science, Naval Postgraduate School, Monterey, CA 93943, USA*

The main objective of this short paper is to describe the relationship between software maintenance and logic programming (both declarative and procedural), and to show how ideas and methods from logic programming (in particular, methods invented by M. Gelfond) can be used in software maintenance. The material presented in this paper partly appeared in (Luqi and Cooke, 1995). The main difference is that (Luqi and Cooke, 1995) is aimed mainly at software engineers, so it only briefly touches on the software engineering problems, while describing in great detail the basics of logic programming. In contrast, in this paper, we assume that the corresponding logic programming notions are well known, but describe the corresponding software engineering applications in greater detail.

1. The problems of software maintenance: brief introduction (why, how, and when)

Software maintenance: why? Real-life programs sometimes produce undesirable results, and sometimes do not run at all. One of the main objectives of software engineering is to ensure that programs do exactly what we want them to do. One of the main ideas in achieving this objective is the idea of *formal specifications*, according to which, we formulate the program requirements (specifications) in precise (*formal*) terms, and try to come up with *verified* programs, i.e., programs for which we can *prove* that the program satisfies these specifications.

In the *ideal* situations, programs exactly follow specifications, and they are as reliable as detailed mathematical proofs. The only thing that can possibly go wrong in such situations is that the wrong program may be (incorrectly) proclaimed to be correct because there was an (accidental) error in the proof. In this case, we need to replace the wrong program by the correct one. Such corrections are called *corrective* software maintenance.

From this "ideal" viewpoint, when we change our specifications, it is like changing the formulation of the theorem: we have to find a proof for the modified theorem, and we have to find the a program that satisfies the new specifications.

In real life, when the specifications are changed, we do not have to design pro-

* Research sponsored by the AFOSR under contract F49620-93-1-0152; the NSF under grant numbers CCR-9058453 and CDA-9015006; by the ARO under grant number ARO-145-91, and NASA NCCW 0089.

grams completely anew. Most often, the existing program can be *modified* so that it satisfies the new specifications. Not only *can* the program be modified, it *should* be modified (rather than written anew), because such a modification (i.e., *re-use* of the existing code) is often the only way to produce the reliable software product under limited human and time resources (see, e.g., [2,11]). These specification modifications and the corresponding program modifications are also called *software maintenance*.

Software engineers usually distinguish between two types of specification modifications:

- *Perfective* modifications occur, e.g., when we want to compute the solution to the given equation with a better accuracy than the existing program provides.
- *Adaptive* modifications occur when the environment changes. For example, in order to design a space mission, i.e., to choose a trajectory, the flight length, etc., we must use the parameters of the spaceship such as its weight, fuel capability, etc. When we upgrade the Space Shuttle (i.e., change these parameters), then we need to modify the program. In software engineering, this necessity to constantly adapt the program to an everchanging context is well recognized; it is, e.g., explicitly stated in Lehman's laws [10] ("For a software system to survive it must evolve").

Comment. Not every program needs adaptive maintenance. Lehman actually classifies programs into S-type programs whose specifications are not likely to change over time (e.g., matrix multiplication or any numerical package) and E-type programs whose specifications change over time. Large software systems usually contain programs of both types, so adaptive maintenance is necessary.

Software maintenance: how? If the necessity to modify the software is recognized, we face an important (and often difficult) problem of actually modifying the software in the right way.

Software maintenance: when? The word "if" in the above paragraph actually hides a crucial problem: how do we know that it is time for modifying software?

This problem is relatively easy for programs that solve well-defined problems for a reasonably simple environment. For such programs, modifications are rare (how often do we need to increase the accuracy of a numerical method?) and the need for such modifications is easily recognizable.

For programs that operate in a more complex environment, the situation is much more complicated. The environment is constantly changing, and programs are usually designed in such a way that they still work correctly under (sufficiently) small changes: for example, an operating system, usually, does not need maintenance if we simply add one more workstation to the net. However, as the small changes accrue, we may eventually arrive at a situation where the old program does not work correctly anymore. Currently, the *user* decides when the maintenance is needed: e.g., the user detects that the system is not performing the way it should (e.g., that there is an error in a result), or that there has been some drastic change in the software environment

or context that requires that the system be modified (i.e., there is a need for adaptive maintenance). Since the accrued changes are minor, it is very difficult to detect the exact moment when the old software is no longer correct. However, if we do not detect this moment exactly, if we wait until the program starts misbehaving, we may get disastrous consequences, such as the explosion of the European Ariane 5 satellite launcher on its June 4, 1996 maiden flight. The guidance software used on this flight was originally designed for the the previous (slower) system Ariane 4; it worked well for several (faster) modifications of Ariane 4, but with the upgrade to Ariane 5 the flight became too fast for the guidance software to handle.

A major problem is, therefore, to *detect* when the maintenance is needed.

This detection problem can be informally reformulated in logical terms: If the program was initially designed exactly according to some specifications, then the need for maintenance means that there is an *inconsistency* between the (old) requirements on which the program was based, and the new specifications that were added later. In these terms, the problem is to *detect inconsistencies*.

2. Logic programming: an appropriate formalism for solving software maintenance problems

Traditionally, classical (*monotonic*) *logic has been used in software engineering.* Traditionally in software engineering, specifications were described in terms of classical (first order) logic. This is done either *directly* in terms of this logic, or *indirectly*, i.e., in terms of some specific formalism whose semantics are defined in terms of first order logic.

In this formalism, a specification for a program consists of one or several logical statements that describe the possible pairs (x, y), where x in an input, and y is an output. If we want to add a new requirement to the system, this means that we have to add a new formula that the pairs must satisfy. The more formulas we add, the fewer pairs will satisfy all of them. In other words, the system, as described by the first order logic, is *monotonic* in the following precise sense: Let us denote the set of all pairs (input,output) that satisfy the given set of formulas S by $p(S)$. Then, if we *increase* the set of formulas, i.e., go from the original set S to a larger set $S' \supset S$, we thus *decrease* the set of all possible pairs: $p(S') \subseteq p(S)$.

Actual specifications are often non-monotonic. Modifications to the program specifications are indeed formulated mostly in terms of additional statements. However, in real life, these statements may not necessarily mean new restrictions on the output; sometimes, these new statements describe new exceptions to the previously formulated requirements, exceptions that were not known before. When an additional statement is a new exception, then the new set $f(S')$ of possible pairs (input,output) is not *smaller*, but *larger* than the original set of pairs $f(S)$: $f(S') \supseteq f(S)$.

Usually, additional requirements consist of statements of both types: some of these statements bring new restrictions, some limit the previous restrictions. As a result,

we often get "incomparable" sets $f(S)$ and $f(S')$, i.e., sets for which neither $f(S)$ is contained in $f(S')$, nor $f(S')$ is contained in $f(S)$ ($f(S) \not\subseteq f(S')$ and $f(S') \not\subseteq f(S)$).

In all these cases, the mapping f is *non-monotonic*. *Non-monotonic* logics (see, e.g., [12–14]) can be useful in formulating the logic behind such mappings.

Logic programming seems to be the most appropriate non-monotonic logic for software engineering. We believe that logic programming is the most adequate logical formalism for describing software maintenance, for the following reasons:

- First, many software requirements are formulated in terms of common sense, and logic programming seems to be a natural and adequate description of commonsense reasoning. For example, logic programming provides a clear distinction between *immutable* specifications (like "Bill and Sam are brothers") that are absolutely true, and *mutable* specifications that are typically true, but may have exceptions (like "It may be assumed/believed that Bill and Sam are kind to each other"). Immutable specifications are described by absolutely true facts and rules like

  $$brothers(bill, sam) \leftarrow,$$

  while mutable specifications can be described as defaults like

  $$kind\_to\_each\_other(bill, sam, S) \leftarrow not\, abnormal\_situation(S).$$

- Second, logic programming is not only a declarative (specification) language. Logic programming, in its Prolog form (and its variants), is also a reasonably efficient *procedural* language. Therefore, if we formulate our restrictions on the pairs (input,output) in terms of a logic program, we not only get a good *formalization* of our specifications, but, when we apply a Prolog compiler to this specification, we may, in many cases, actually get a good *prototype implementation*, that, given an input, produces an output that is consistent with these specifications.

3. How to describe inconsistencies?

Reminder: inconsistencies need to be described. Logic programming in general (and Prolog in particular) helps in the design of a program that satisfies given specifications and thus, helps to solve the problem of *how* to maintain software.

However, as we have already mentioned, there is another problem that is, often, even more important: to find out *when* it is necessary to change the software, i.e., in logical terms, when adding a new requirement leads to an inconsistency. How can we do it?

Prolog-type logic programming is not sufficient to describe inconsistencies. In traditional (first order) logic, inconsistency means that for some query q, we can conclude both q and $\neg q$. In "standard" logic programming, there is no direct analog of inconsistency: the only negation available is negation as failure, according to which "not q"

is deduced if and only if q cannot be deduced. This definition automatically excludes any possibility of an inconsistency.

Logic programs with classical negation. To describe inconsistency, we therefore need to add another (more classical) negation operation to the "standard" logic programming formalism. Such an addition was proposed by M. Gelfond and V. Lifschitz in [7,8] (programs that use thus defined classical negation are called *extended logic programs*).

Extended logic programming, with a classical negation $\neg$, can indeed describe inconsistency. For example, let $des(In, Out)$ be the predicate that describes the desired output Out for a given input In, and suppose that one of the requirements is that the Out should be uniquely determined by the input In. This requirement can be formulated as follows:

$$\neg des(In, Out') \leftarrow des(In, Out), Out \neq Out'.$$

Then, if the logic program allows two different outputs (e.g., $Out = 9$ and $Out' = 10$) for the same input (e.g., for the list $In = [4, 5]$), i.e., if we can conclude both $des([4, 5], 9)$ and $des([4, 5], 10)$ from this program, we get $\neg des([4, 5], 9)$ and thus, inconsistency.

How to practically detect inconsistencies? Since classical negation is useful for detecting inconsistencies, it is desirable to be able to incorporate classical negation into Prolog. In their original papers, M. Gelfond and V. Lifschitz recommended the following incorporation: for every predicate p whose classical negation $\neg p$ is in the program, we replace $\neg p$ by a special new predicate np. This replacement actually takes place as a result of the definition of an *answer set* for an extended logic program (from [7,8]): namely, an answer set is defined as a *stable model* [6] for the replaced program if this stable model does not contain p and np at the same time, and *all* atoms if the stable model contains such a contradictory pair.

If we do such a replacement, then, informally, to get an answer to the query "p?", we can ask two questions: "p?" and "np?". Then, depending on the answers to these questions, we get four possibilities:

- If the Prolog's answer to p is "yes" and to np is "no", then the answer to the original query is "yes".
- If the Prolog's answer to p is "no" and to np is "yes", then the answer to the original query is "no".
- If the Prolog's answer to both p and np is "no", then the answer to the original query is "unknown".
- If the Prolog's answer to both p and np is "yes", then the answer to the original query is "inconsistent".

This algorithm is not completely in line with the standard semantics of classical negation. The above algorithm, however, is not exactly in line with the answer set semantics for extended logic programs. Indeed, if we take a consistent logic program,

e.g., a fact $p \leftarrow$, and add to it a pair of inconsistent sets $q \leftarrow$ and $\neg q \leftarrow$, then the resulting extended logic program

$$p \leftarrow$$
$$q \leftarrow$$
$$\neg q \leftarrow$$

is inconsistent. Hence, from the viewpoint of the answer set semantics of extended logic programs, its only answer set is $\{p, \neg p, q, \neg q\}$, and the answer to the query p should be "inconsistent". However, if we simply replace $\neg q$ by nq, we get a logic program

$$p \leftarrow$$
$$q \leftarrow$$
$$nq \leftarrow$$

for which the answer to the query "p?" is "yes", and to the query "np?" is "no", which, according to the above heuristic, means that the answer to the original query "q?" is "yes" (and not "unknown" as the answer set semantics implies).

So, even for this simplest case of inconsistency the above algorithm is not in line with the semantics. There are two possible ways of dealing with this problem:

- we can modify the algorithm, or
- we may use some modification of the semantics.

In principle, it is possible to modify the above algorithm so that it would be more in line with the answer set semantics: For example, we can add a new predicate $incon$ and a new "meta-rule"

$$incon \leftarrow P(X), \neg P(X);$$

then, in order to get the answer to a query p, we ask not two but *three* queries: "p?", "np?", and "$incon$?", and return the answer "inconsistent" whenever the Prolog's answer to $incon$ is "yes". However, with this proposal, we, in effect, make the Prolog compiler seek the answers to *all* possible queries even when we are interested in the value of only *one* predicate. This will drastically increase the running time of the Prolog program and, for large logic programs, make the idea unrealistic.

Gelfond's modification of answer set semantics is most appropriate for the description of software maintenance. Since trying to be as close to answer set semantics as possible means a drastic increase in running time, we believe that it is preferable to keep the original algorithm but to modify the *semantics* instead. Such a modification was actually proposed by M. Gelfond himself (unpublished): If we are given an extended logic program $\mathcal{P}$, we can define its *answer set* as a *stable model* of the program $\mathcal{P}'$ that is obtained from $\mathcal{P}$ by replacing each classical negation $\neg p$ with a

new predicate np even if the resulting stable model contains a "contradictory" pair p and np.

For example, if we apply this substitution to the above simple inconsistent extended logic program $\mathcal{P}$, then the resulting logic program $\mathcal{P}'$ (without classical negation) has exactly one stable model $\{p, q, nq\}$. Therefore, in the modified semantics, the above simple inconsistent program $\mathcal{P}$ has exactly one answer set $\{p, q, \neg q\}$, so the algorithm's answer "yes" to the query "p?" is exactly what this semantics predicts.

We have used the above algorithm and the corresponding semantics to check the inconsistency of several realistic specifications (see, e.g., a missile firing example described in [11]). These initial successes of using the simplest logic programming tools make us believe that the applications of more complicated and more realistic logic programming formalisms and methods to software engineering will be fruitful (provided, possibly, that additional research is done on the above modified semantics of answer sets and on its unpublished extensions to epistemic specifications).

Remark: it is not necessary to ask two queries. In the above algorithm, to answer a query "p?", we have to run Prolog twice (for "p?" and for "np?"). It is relatively easy to automate this "doubling" inside Prolog itself (this is how we actually dealt with the applications). To do that, we can introduce the new predicates `notl(.)` and `ans(.,.)`, and add the following rules to the original Prolog program:

```
notl(P):-P,!,fail.
notl(P).

ans(P,true):-P,notl(not(P)).
ans(P,false):-not(P),notl(P).
ans(P,inconsistent):-P,not(P).
ans(P,incomplete).
```

Here, `notl(P)` stands for negation as failure (and one can easily trace that the query "`notl(P)`?" returns "yes" if and only if the query `P` returns "no"), `not(P)` stands for classical negation (it has to be specified by rules from the database), and `ans(P,A)` returns one of the four possible answers `A` ("true", "false", "inconsistent", and "incomplete" meaning "unknown") to the query `P`.

For example, if we have both `des([4,5],9)` and `des([4,5],10)` in the logic program, and we have a rule (as above) that leads from `des([4,5],10)` to the classical negation

```
not(des([4,5],9)),
```

then Prolog's answer to the query

```
ans(des([4,5],9),A))
```

(or to a more general query `ans(des([4,5],X),A))`) is that `A` is "inconsistent".

Checking completeness: an additional advantage of logic programming approach. The logic programming approach not only helps us to solve the important software engineering problem of checking the *consistency* of specifications, but it also helps with the equally important problem of checking whether the specifications are *complete* (the importance of checking completeness is emphasized in [4,5]).

Normally, the set of specifications is assumed to be complete. In logic programming terms, this means that we are dealing with the logic programs that have a unique answer set. The class of logic programs with unique answer sets is large [9]. For example, logic programs that do not contain negation as failure at all, and programs in which no negation as failure is contained in a loop (in particular, *stratifiable programs* [1,3]) are guaranteed to have a unique answer set. We believe that all specifications that are sufficiently complete to be considered for the purposes of a software development project, correspond to a logic program with a unique answer set.

However, it is almost certain that the initial versions of the specifications for any real system will not be complete in this sense. It is therefore desirable to design a Prolog-based answering mechanics that would provide correct answers to queries even if the program has several answer sets (i.e., if specifications are not complete).

Acknowledgements

Thanks to friends and colleagues, Valdis Berzins, Michael Gelfond, Joseph Goguen, and Vladik Kreinovich for extensive comments and suggestions. Thanks also to the excellent reviewers who gave prompt and excellent guidance.

References

[1] K. Apt and H. Blair, Arithmetic classification of perfect models of stratified programs, Fundamenta Informaticae 13 (1990) 1–18.
[2] V.R. Basili, Viewing maintenance as reuse-oriented software development, IEEE Software 7(2) (1990) 19–25.
[3] C. Baral and M. Gelfond, Logic programming and knowledge representation, Journal of Logic Programming 19 (1994) 73–148.
[4] D. Cooke, A. Gates, E. Demirors, O. Demirors, M. Tanik and B. Kraemer, Languages for the specification of software, Journal of Systems and Software 32 (1996) 269–308.
[5] A.M. Davis, A comparison of techniques for the specification of external system behavior, Communications of ACM 31(9) (1988) 1098–1115.
[6] M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, in: *Proc.* 5*th International Conference and Symposium on Logic Programming*, Seattle, Washington (August 15–19, 1988), eds. R. Kowalski and K. Bowen, pp. 1070–1080.
[7] M. Gelfond and V. Lifschitz, Logic programs with classical negation, in: *Proceedings of 7th International Conference on Logic Programming* (Jerusalem, 1990) pp. 579–597.
[8] M. Gelfond and V. Lifschitz, Classical negation in logic programs and deductive databases, Journal of New Generation Computing 9(3,4) (1991) 365–387.
[9] M. Gelfond and H. Przymusinska, Stratified extended logic programs. Draft copy of a paper in preparation.

[10] M. Lehman, Programs, life cycles, and laws of software evolution, Proceedings of the IEEE 68(9) (1980) 1060–1075.
[11] Luqi and D.E. Cooke, How to combine nonmonotonic logic and rapid prototyping to help maintain software, International Journal of Software Engineering and Knowledge Engineering 5(1) (1995) 89–118.
[12] C.V. Ramamoorthy and D. Cooke, The correspondence between methods of artificial intelligence and the production and maintenance of evolutionary software, in: *Proceedings of the Third International IEEE Conference on Tools for Artificial Intelligence* (November, 1991) pp. 114–118.
[13] C.V. Ramamoorthy, D. Cooke and C. Baral, Maintaining the truth of specifications in evolutionary software, International Journal of Artificial Intelligence Tools 2(1) (1993) 15–31.
[14] J.-P. Tsai and T. Weigert, A knowledge-based approach for checking software information using a non-monotonic reasoning system, Knowledge-Based Systems 3(3) (1990) 131–138.