Identification and Classification of Inconsistency in Relationship to Software Maintenance*

Daniel Cooke[1], Luqi[2], and Vladik Kreinovich[1]

[1]Computer Science, University of Texas at El Paso, El Paso, TX 79968
[2]Computer Science, Naval Postgraduate School, Monterey, CA 93943

Abstract

This paper provides an overview of the relationship between recent work in logic programming and recent developments in software engineering. The relationship to software engineering is more specifically concerned with how formal specifications can be used to explain and represent the basis of software maintenance and evolution. Some of the results reviewed here have appeared in [5] and [13]. These previous results are summarized, extended, and made more general in this paper.

1 Introduction

Maintenance activities can be divided into three distinct classes: corrective, perfective, and adaptive. Corrective maintenance largely reflects the failure of software engineers to validate and verify software specifications and programs with respect to software specifications, respectively. Perfective maintenance is traditionally viewed as a form of maintenance necessary to improve or change the performance of a system, but not its functionality. Adaptive software maintenance alters the functionality of a system to reflect a changing software context. Even if software is valid and verified, adaptive changes continue to be necessary.

Currently, the need for software maintenance is detected by the software user. In the most general case, the user detects that the system is not performing the way it should (i.e., there is a need for perfective maintenance), there is an error in a result (i.e., there is a need for corrective maintenance), or there has been some change in the software environment or context that requires that the system be modified (i.e., there is a need for adaptive maintenance). In all of these cases the system is behaving in a manner that is *inconsistent* to the user's expectation. At the heart of the need for maintenance is the problem of inconsistency.

But what exactly is maintenance and what is software evolution? Without a formalism it is difficult to claim that anything is truly understood. Without a formalism, one must rely on his/her instinct and intuition. In this paper, we focus on how the maintenance activity can be understood using recent developments in Logic Programming Research. These results and their relationship to software maintenance are key to understanding what software maintenance actually entails. Logic programming results provide a formal backdrop to an explanation and understanding of software evolution and software maintenance. In this paper software evolution is to be understood as the changing functionality of live and useful software. Consequently, the focus of this paper will be on adaptive maintenance.

*Research sponsored by the AFOSR under contracts F49620-93-1-0152 and F49620-95-1-0518; by the NSF under grant numbers CCR-9058453, CDA-9015006, EEC-9322370, and DUE-9750858; by the ARO under grant number ARO-145-91, and by NASA under grant number NCCW 0089.

$$
\begin{array}{|ccccccccc|}
\hline
C_0 & \rightarrow & C_1 & \rightarrow & C_2 & \rightarrow & \ldots & \rightarrow & C_n \\
\downharpoonleft & & \downharpoonleft & & \downharpoonleft & & & & \downharpoonleft \\
S_0 & & S_1 & & S_2 & & \ldots & & S_n \\
\downharpoonleft & & \downharpoonleft & & \downharpoonleft & & & & \downharpoonleft \\
P_0 & & P_1 & & P_2 & & \ldots & & P_n \\
\hline
\end{array}
$$

Figure 1: Context Changes Trigger Software Changes.

2 Adaptive Changes: Software Maintenance and Evolution

Adaptive changes result from a changing software context. By context (or software context), we mean the parts of the real world with which a specified system is to interact, including people and organizations as well as other programs, databases, and hardware devices. When valid, an initial specification S_0 of a software system reflects the initial context C_0 as well as the appropriate ways in which the software system is to interact with C_0. In an ideal software development process, only those parts of the context which will truly affect the software system should be represented by the software specification - all irrelevant features of the context are abstracted out of S_0. The initial program P_0, if correct with respect to S_0, should operate correctly in C_0. Whether or not a specification reflects the context is a problem of validation, and the correctness of a program with respect to a specification is the problem of verification.

In practice, one often finds that S_0 does not correctly reflect C_0 (i.e., the specification is invalid) and P_0 is not correct with respect to S_0 (i.e., the program does not completely satisfy the specification). In such cases, corrective and/or perfective changes are needed.

In any event, the context of a typical system will change. Over the life of the system, we typically observe a series of contexts C_i which should lead to a corresponding series of specifications S_i and a corresponding series of programs P_i, as shown in Fig. 1.

In this ideal case, the need for adaptive change becomes more obvious. If the context is C_i (where $i > 0$) and the current version of the program is P_{i-1}, then the program does not correspond to a valid specification. It is clear that each change in the software context requires an appropriate adaptation of the software. As Lehman's laws [11] indicate, for a software system to survive it must evolve; it must adapt to its everchanging context. We call the change in context from C_i to C_j (where $j > i$) a *context shift*, denoted by $C_i \rightarrow C_j$ in Figure 1. Figure 1 also illustrates the notion of a mapping from a context to a specification and from a specification to a program, denoted as: $C_i \downharpoonleft S_i$ and $S_i \downharpoonleft P_i$. A context shift should result in maintenance activity where valid changes are made to a specification and correct changes are made to a program.

Software evolution should be viewed as a continual process of re-validation and re-verification. When the context changes, the old system is at best, correct with respect to an invalid specification. The context shift recommends the need to re-validate the specification and then re-verify the program. In fact, the software process model presented in figure 1 indicates that software maintenance should not be viewed as an activity distinct from software development. The difference between initial development and subsequent maintenance is a matter of the degree to which specifications and associated software are reused. The maintenance process suggested here is is similar to that found in Basili's Full Reuse Maintenance Process [2].

The contribution made in this paper is the use of a logic which can be used to formalize the causes of adaptive maintenance. As a consequence of the foregoing discussion, one should realize that the logic is to indicate when a specification is no longer valid.

3 Specification Validation

In order to be valid a specification must be *complete, consistent,* and *precise* [4, 6]. A precise specification means that there is no ambiguity in the statement of the specification. In order to satisfy this requirement, a specification must be expressed in a language for which there is a formal (or mathematical) semantic. Expressing the specification in logic satisfies this requirement.

In order to be complete, a specification must correctly express all of the functionality desired in the software system. Consistency means that there are no conflicting definitions in the specification. The reader shall see that the extended logic programming semantic [7] provides for a clear representation of completeness and consistency.

4 Modeling a System and its Context

A system specifier attempts to make precise statements about the intended behavior of the system and its context. There are two important classes of specification in any system: those which are *immutable* and those which are *mutable*. Immutable specifications are statements about the software and/or its context which remain true for all time. A mutable specification is a statement which is believed or assumed to be true, i.e., an assumption or a belief. Beliefs or assumptions are typically true in some contexts but not in all possible contexts.

EXAMPLE 1. Immutable Specification: Bill and Sam Cooke are brothers.

EXAMPLE 2. Mutable Specification: It may be assumed/believed that Bill and Sam are kind to each other.

The knowledge that Bill and Sam are brothers is a known fact which is forever true (i.e., immutable). They are now and will forever be brothers. However, it is a belief (i.e., mutable) that Bill and Sam are kind to each other. The validity of this belief can change with time. With alarming frequency, Sam and Bill may substantiate or invalidate this statement through their behavior.

The notions of immutable and mutable specifications are analogous to Lehman's S- and E-type programs. The types are based upon the character of the program's specification. An S-type program is a program which is correct with respect to specifications which do not change over time while an E-type program is one which is correct with respect to specifications which may indeed change over time. A large system is typically comprised of some mixture of the program types.

An E-type program has to evolve because the validity of the assumptions coded into the program change with time. The assumption in an E-type program is a mutable specification. The mutable specifications serve as the seed of adaptive maintenance. The mutable specifications correspond to the parts of the software context which are susceptible to the changes which result in context shifts.

Results from the study of nonmonotonic logic serve as a basis for understanding the mutable specification [14, 15, 16]. Nonmonotonic logics provide formalisms to handle beliefs (or assumptions). Intuitively, nonmonotonic logics allow the retraction of beliefs when new information, which contradicts those beliefs, is presented. In contrast, in monotonic logics, once the truth of a statement is established, new information cannot invalidate the justification for believing the statement (i.e. its proof or derivation).

Consider the following definitions of monotonic and nonmonotonic. Let S and S' represent a specification and a changed specification, respectively. In predicate logic, a specification is a set of assertions (both extensional and intensional). Let $f(S)$ and $f(S')$ represent the interpretations of S and S'. In other words, $f(S)$ and $f(S')$ are specified relations of S and S'. Specified relations can be viewed as a listing of all possible stated and derived assertions. For example, suppose a specification S is given in Prolog as:

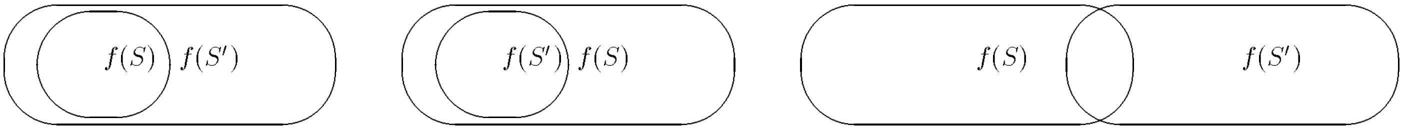
```
p(0,1).
p(N,X):-N > 0,N1 is N - 1,p(N1,Y), X is N * Y.
p(N,error):- N < 0.
```

The specified relation $f(S)$ is:

$f(S) = \{..., p(-2, error), p(-1, error), p(0,1), p(1,1), p(2,2), p(3,6), p(4,12), ...\}$

If the specification is valid, the specified relation relates each input of a program to all valid output values. The most important observation to make is that $f(S)$ is a model of the software context. Therefore, if S is valid, then a change from $f(S)$ to $f(S')$ represents a model of a software context shift: $C_i \rightarrow C_{i+1}$.

Def. A function f is *monotonic* if and only if $\forall S, S'(S \subseteq S' \Rightarrow f(S) \supseteq f(S'))$. (Where $\Rightarrow$ denotes implication.) □



$f(S) \subseteq f(S')$ $\qquad$ $f(S) \supseteq f(S')$ $\qquad$ $f(S) \not\subseteq f(S') \wedge f(S) \not\supseteq f(S')$

Figure 2: Relationships among specifications due to change.

The definition of nonmonotonic negates the formula above:

Def. f is *nonmonotonic* if and only if $\exists S, S'(S \subseteq S' \wedge f(S) \not\subseteq f(S'))$. □

The definition of nonmonotonic suggests a classification of specification changes where the classification is based on the fact that we can add to or delete from S, i.e., $S \subseteq S'$ or $S \supseteq S'$. There are exactly three ways that the addition to or deletion from S impacts the specified relation. The three ways are based upon the possible relationships between $f(S)$ and $f(S')$ due to $C_i \rightarrow C_{i+1}$, assuming that no change will result in $f(S) \bigcap f(S') = \emptyset$.

1. $f(S) \subseteq f(S')$;
2. $f(S) \supseteq f(S')$;
3. $f(S) \not\subseteq f(S') \wedge f(S) \not\supseteq f(S')$;

Through these relationships one immediately sees the primitive effects of a context shift $C_i \rightarrow C_{i+1}$. Observe that in all of the cases for change in the specified relation, there will exist an inconsistency between the old and the new specified relation. Suppose it is possible to know the information about both S and S'. This is actually what the software user possesses when he/she detects an inconsistency: $f(S)$ is what the software produces and $f(S')$ is what the user expects. *Assuming p is an input/output relation, a change to the specification $S \neq S'$ is detectable through the detection of an inconsistency:*

1. $f(S) \subseteq f(S')$ (and $f(S) \neq f(S')$) means $\exists p(p \notin f(S) \wedge p \in f(S'))$;
2. $f(S) \supseteq f(S')$ (and $f(S) \neq f(S')$) means $\exists p(p \in f(S) \wedge p \notin f(S'))$;
3. $f(S) \not\subseteq f(S') \wedge f(S) \not\supseteq f(S')$ means $\exists p, p'(p' \neq p \wedge (p \notin f(S) \wedge p \in f(S')) \wedge (p' \in f(S) \wedge p' \notin f(S')))$.

Figure 2 presents the meanings of the three possible relationships between $f(S)$ and its successor $f(S')$ in terms of Venn Diagrams. Absent is a diagram for disjoint sets, $f(S) \bigcap f(S') = \emptyset$. This is due to the fact that disjoint sets represent completely different systems.

The facility to have knowledge about S and S' requires the ability to monitor relevant aspects of the corresponding software contexts. Implied by the foregoing is a framework for the detection of the need for adaptive maintenance. The framework is as follows:

1. The specifier of a problem solution must identify the specifications that are actually assumptions (i.e., those specifications that are mutable);
2. The specifier must determine how the assumptions can be contradicted or, in other words, lead to an inconsistency (this leads to the rules by which it is possible to detect context shifts);
3. The specifier must identify the additional inputs necessary to detect the context shifts; and
4. The system must have the facility to detect inconsistencies when they arise.

5 Overview of Context Dependent Specifications

5.1 Extended logic programming as a natural basis for a specification language

There is a need for a (formally defined) specification language. In order to be able to *automatically* check consistency and completeness of different specifications, i.e., to make a *computer* do this checking, we must describe these specifications in a language that a computer can understand. In other words, we need a *formal* language for describing specifications.

First attempt: the language of first order logic. Specifications (i.e., properties that the designed software must satisfy) can be arbitrarily complicated. In particular, these specifications may include terms like "and", "or", "not", "if ... then", "for every x", "there exists an x", etc., which are naturally formalized as logical *connectives* and *quantifiers*. Therefore, a natural idea is to allow, as specifications, arbitrary formulas that can be obtained from some basic formulas by using these connectives $\wedge$, $\vee$, $\neg$, $\rightarrow$, and quantifiers $\forall x$ and $\exists x$. The resulting formulas are called formulas of *first order logic*.

The main drawback of using first order logic. First order logic describes arbitrarily complex specifications very well; for example, it is the basic language used in the formalization of concepts in mathematics. If our goal was simply to *formalize* knowledge, then first order logic would be a perfect solution.

However, our goal is not simply to *formalize* the specifications, but also to *use* these formalized specifications as an input to a program that will check whether these specifications are consistent and/or complete. The existence of such a program means that we must have an *algorithm* that would, e.g., check whether a given set of formalized specifications is consistent or inconsistent.

For this purpose, first order logic is not a very adequate language, because it is well known that no algorithm is possible for checking consistency of first order formulas (this impossibility is a direct corollary of the famous Gödel's theorem).

Logic programming: a way to make logic formulas algorithmic. Gödel's result shows that the main drawback of first order logic as a specification language is that the language of the first order logic is not algorithmic. It is therefore desirable to modify this *logical* language and make it algorithmic, i.e., make it suitable for *programming*. Such logic-based programming languages have indeed been invented; the first and most used of these languages is *Prolog* (this name is an abbreviation of *Pro*gramming in *Log*ic).

Basic logic programming: motivation and description. One of the main ideas behind making logic algorithmic is as follows: since allowing *all* logical connectives and quantifiers makes the problems non-algorithmic, we should allow only *some* of them. Which of the connectives and quantifiers are the most important for representing specifications?

A typical specification describes what the program should do in different cases: e.g., if something happens, then it should perform a certain action, etc. Specifications are rarely formulated in terms of negative conclusions or disjunctions (do this *or* that). Therefore, the most important connective for knowledge representation is an *implication* (if – then). In view of this, basic logic programming only allows implications. Moreover, since combinations of connectives (like $(A \rightarrow B) \rightarrow C$) can become extremely complicated and hard to understand, the designers of the basic logic programming language allowed only single implications, but not their combinations. As a result, a typical program in basic Prolog consists of statement of two types:

- *facts* A, i.e., statements that do not contain any logical symbols at all; facts are also known as *elementary* statements, or *atoms*; and
- simple "if"-"then" rules of the type

  $$\text{"if } A_1, \ldots, \text{ and } A_n, \text{ then } B\text{"},$$

  where A_i and B are atoms (elementary statements).

How can we formally describe these rules?

It is possible to describe these rules in first order logic, where, e.g., the above statement can be written as

$$(A_1 \& \ldots \& A_n) \rightarrow B.$$

However, specifications are not always given in terms of logic programming: we often start with specifications written in first order logic, and then translate these specifications into the language of logic programming. In view of this, it is desirable to distinguish between the original (non-algorithmic) statements from first order logic and the resulting (algorithmic) statements which form the logic program. To make this distinction easier, researchers in logic programming use a re-oriented arrow to denote implication; for example, for the above "if"-"then" rules, they use the notation

$$B \leftarrow A_1, \ldots, A_n.$$

For *facts*, the distinction is not that critical, but sometimes, we also need to distinguish between the elementary statements from the original logical specification and the facts from the resulting logic program. One of the ways to make this distinction easier is to represent each fact A as a rule with an empty right-hand side, i.e., as a rule

$$A \leftarrow$$

So, a *basic* logic program is simply a collection of rules and facts.

Basic logic programming: a natural choice of semantics. The main objective of our formal representation of a specification is to be able,
GIVEN:

- a specification F and
- a property Q that we want to check, to

RETURN: an answer describing whether programs satisfying this specification also satisfy the property Q.

The description of what answers the system should return for each formal specification (and for each queried property) is called the *semantics* of the specification language.

For basic logic programs, semantics immediately follow from the fact that these programs are actually a particular case of formulas of first order logic. In general, if specifications are described by an arbitrary first order formula F, then for every queried property Q, we have one of the following three situations:

- The property Q follows from the specifications F; in this case, this property Q is *true* for every program that satisfies these specifications (or, in logical terms, in all *models* of this formula F).
- The negation $\neg Q$ of the property Q follows from the specifications F; in this case, this property Q is *false* for every program that satisfies these specifications (i.e., in all *models* of the formula F).
- Neither the property Q, nor its negation follow from the specifications F; in this case, this property is *true* for some programs (models) that satisfy these specifications but *false* for the other programs (models) that also satisfy the same specifications F.

  For example, if we specify the *results* of a program but do not restrict its running time, and then take as Q some time limitation (e.g., that the program will take no longer than a minute to run), then it is quite possible that some program produces the correct result for an under-a-minute time, while other programs, that also produce correct results (and thus, also satisfy the same specifications), will take much longer than a minute to run.

(Of course, theoretically, there is a fourth possibility: that the specifications F are inconsistent, so no program can satisfy *all* the specifications that we have combined into a formula F.)

This classification can be also applied to rules and facts that form a basic logic program. Fortunately, basic logic programs are a particularly *simple* case of general first order formulas, and due to this simplicity, for first order formulas, we get a simplification of this classification. Indeed, all facts and rules that form a basic logic program remain true if we simply consider a model in which all elementary properties are true. Therefore, whichever of these properties Q we ask about, it is always possible that this property is true. In other words, for basic logic programs, instead of the above *three* possibilities, we have only *two* possibilities:

- First, it is possible that the property Q is *true* for all models of this logic program.
- Second, it is possible that in some models of the logic program F, this property Q is *false*.

The actual Prolog compiler, given a logic program F and a query Q, decides which of these two cases holds. Of course, it makes no practical sense to let the compiler return either of these two *long* messages that describe the corresponding cases. Therefore, only the *shortened* messages are returned:

- In the first case, the compiler returns the shortened message "true" (or, even shorter, "yes");
- in the second case, it returns a shortened message "false" (or, even shorter, "no").

Exception handling and generalized logic programs. Basic logic programs capture many important specifications, but there is one important feature of specifications that these simple formulas have difficulty capturing: exception handling.

Many specifications are described in terms of *exceptional* situations: e.g., a natural specification for a tax program would describe all exceptions to normal taxes (disability, spending most of the year abroad, etc.), and then say that unless one of the exceptions applies, taxes should be computed using the given formula. Exceptions themselves can be captured easily by simple implications of the type used in basic logic programs, e.g., "if a person has spent 12 months abroad, then he or she does no pay any taxes". The difficulty appears when we try to describe a statement about a *normal* situation ("unless one of the exceptions applies ...").

It is, of course, potentially possible to describe this statement as a normal if-then statement, by explicitly enumerating all the exceptions in the conditions of this rule. However, there may be very many exceptions (and moreover, many rules may have different exceptions), and the enumeration of these exceptions would drastically increase the size of the logic program, and thus, inevitably increase the time that is necessary to answer queries.

To avoid these complications, researchers in logic programming proposed the notion of a *generalized logic program*, also called *logic program with negation as failure*. The idea of negation as failure can be explained using the above tax example: We have some rules that explain what an exception is, i.e., several rules that conclude, based on some assumption, that the given situation S is indeed exceptional. In logic programming terms, these rules have the form

$$exc(S) \leftarrow \ldots$$

We also have a rule in which one of the conditions is "if none of the exceptions apply", meaning "if none of the other rules specify that this is an exceptional case", or, in yet another form, "if we have failed to prove that this is an exception". This "not an exception" is denoted by *not exc* and, since this *not* actually indicates failure to prove, it is called *negation as failure.*

By definition, negation as failure only occurs in the *conditions* of the rules. Thus, a *generalized* logic program can be defined as a collection of facts and rules of the following type:

$$B \leftarrow A_1, \ldots, A_n, not\ C_1, \ldots, not\ C_m,$$

where B, A_i, and C_j are elementary statements.

Semantics of generalized logic programs. For generalized logic programs, we also need to determine a semantic, i.e., we also need to be able to determine, for a given program F, whether a given query Q is true or not.

Negation as failure is not a typical logical connective, and therefore, in contrast to the case of basic logic programs, we cannot directly *deduce* the semantics of a generalized logic program from the known semantics of first order logic. However, we can still deduce this semantic *indirectly*, by checking the *consistency* of the resulting assignment of "true" and "false" to different elementary statements from the program.

Indeed, let us assume that F is a generalized logic program, and that to every elementary statement from this program, we somehow assign "true" or "false". In mathematical terms, this means that we have selected, in the set of all atoms of the original logic program, a subset T formed by those atoms that our semantic deems "true".

This means, in particular, that we have assigned "true" or "false" to every statement about exceptions, i.e., that we know about each situation and each rule, whether this situation is an exception to this rule or not, which means, for each statement C that occurs under negation as failure, we know whether this statement is true or not. In this case, we can determine which rules are applicable and which are not and thus, transform the original rules into new rules that do not contain negation as failure. This transformation can be done as follows:

- If one of the conditions of a rule is *not* C for some atom C that is true, then this rule is not applicable, and we can safely delete it. (Informally, the presence of the condition *not* C means that this rule is only applicable in *normal* situations, in which there is no way to prove C; the fact that C is true means that we have an *exceptional* situation, and thus, the rule is not applicable.)
- If for all exception-type conditions *not* C_i of a rule, C_i is not true (i.e., $C_i \notin T$), then this rule is indeed applicable and therefore, we can simply delete these conditions *not* C_i from the list of conditions. (Informally: this situation is indeed *non-exceptional*, so the rule *is* applicable.)

As a result of this transformation, we get a new logic program without negation as failure. For this new basic logic program, we can use the above-described semantics and find the resulting set $T_{\rm res}$ of true atoms (i.e., of elementary statements that are true according to this transformed logic program). This set should, of course, coincide with the original set T.

This consistency requirement $T_{\rm res} = T$ only holds for *some* sets of atoms T. Sets of atoms for which $T = T_{\rm res}$, i.e., sets that remain stable (do not change) under this transformation from T to $T_{\rm res}$, are called *stable models* of the original logic program.

The natural consistency requirement eliminates some possible models T; this elimination is very efficient, to the extent that for many important classes of logic programs, there is only one stable model. For example:

- A program that does not contain negation as failure at all is guaranteed to have a unique stable model.
- Furthermore, a program may contain negation as failure but as long as these negations do not form a loop, the program is guaranteed to have a unique stable model.
- Since *stratifiable programs* [1] do not contain negative cycles, stratifiable programs also have unique stable models [3].

There are many other classes of programs with unique stable models (see, e.g., [10]). However, there are also logic programs which have two or more stable sets. For example, one can easily check that a simple logic program

$$p \leftarrow not\ q;$$

$$q \leftarrow not\ p;$$

has exactly two different stable models: $\{p\}$ and $\{q\}$. (This program does not fall into one of the above classes because it has a loop of negations: the rule for p contains a negation of q, and the rule for q contains a negation of p.)

Such logic programs with multiple stable models usually describe *incomplete* specifications.

> E.g., in the above example, the information described by these two rules, basically, says that if p is false, then q must be true, and vice versa. In other words, this information says that either p or q must be true, but it does not specify which of these two atoms is true. Naturally, we have *two* possible stable models here: one in which p is true, and another in which q is true.

We believe that for all specifications, that are sufficiently complete to be considered for the purposes of a software development project, the corresponding logic program has a unique stable model. For such logic programs, the answer to the query Q is "true" if Q belongs to the unique stable model, and "false" if Q does not belong to this unique stable model T.

Comments. Ideally, we should have complete specifications and thus, a logic program with a unique stable model. However, *realistically*, it is quite possible that specifications are incomplete (especially on the initial stages of software design). In this case, we face two problems:

- defining the *semantics*, i.e., what is "true" and what is "false"; and
- finding ways to *complete* the incomplete specification.

Semantics are relatively easy to define: if a *generalized* logic program has several possible stable models, then it is natural to return "true" if Q holds in *all* stable models, and "false" otherwise.

On the other hand, *completion* is often a difficult task. A generic automatic completion mechanism would be useful in practice for diagnosing incompleteness and helping developers understand what choices have to be made in order to make their specifications complete.

Extended logic programs. Generalized logic programs represent *positive* knowledge, i.e., facts (e.g., "an atom A is true"), and "if"–"then" rules ("if $A_1, \ldots, A_n$ are true, then B is true"). In addition to *positive* facts and rules, our specification sometimes include *negative* facts and rules: we know that some elementary statement should not be true, or that under some conditions $A_1, \ldots, A_n$, a fact B should not be true. In other words, we must be able to describe *negation* in its classical sense (i.e., not as negation as failure). Generalized logic program do not allow us to describe this type of knowledge: they do have negation, but what they have is a non-standard "negation as failure". So, to describe such negative knowledge, we must be able to add a more traditional, "classical" negation $\neg$ to the generalized logic programs. In logic programs *extended* thus, a general rule is still of the type

$$B \leftarrow A_1, \ldots, A_n, \mathit{not}\ C_1, \ldots, \mathit{not}\ C_m,$$

but now B, A_i, and C_j are no longer always atoms, they may be also (classical) negations of atoms, i.e., statements of the type $\neg E$ for some atom E.

In general, atoms and their classical negations are called *literals*; so, we can say that A, B_i, and C_j are *literals*.

Semantics of extended logic programs. When we move from generalized to extended logic programs, we replace each original atom E with *two* literals: A and $\neg A$. If we were still in classical logic, then there would be no need to consider $\neg A$ separately, because in classical logic, the truth value of $\neg E$ is uniquely determined by the truth value of E (namely, it is exactly the opposite truth value). However, in logic programming, where knowledge may be incomplete, it is quite possible that an atom A is not true, but this does not necessarily mean that its negation is necessarily false: it may simply mean that we do not know whether A is true or not – it is true in some models and false in others.

So, in an extended logic program, even if we know the truth values of all the atoms, this does not automatically enable us to determine the truth values of their classical negations; in this sense, atoms and their classical negations can be viewed as *different* elementary formulas. This fact leads to the following natural semantics of an extended logic programs:

- We treat each atom and its classical negation as two different atoms; as a result, we get a generalized logic program (with possibly twice as many atoms as before).
- We then use the above stable model semantics to determine which of the new atoms are true and which are false.

In this semantic, each *new* atom (i.e., each literal) is either true or false. As a result, for every *original* atom A, we have four different options:

- The atom A is true, while its classical negation $\neg A$ is not true; in this case, we conclude that A is *true*.
- The atom A is not true, while its classical negation $\neg A$ is true; in this case, we conclude that A is *false*.
- Neither the atom A nor its classical negation $\neg A$ are true; in this case, we conclude that the truth value of A is *unknown* (i.e., that our information about A is incomplete).
- There is also a possibility that *both* the atom A and its classical negation $\neg A$ are true; in this case, we conclude that our information about A is *inconsistent*.

This semantic can be described by the following definition:

Def. Let Π be an extended logic program (i.e., a logic program with both negation as failure and classical negation). By an *s-answer set* for Π, we mean a *stable model* of a generalized logic program that is obtained if we treat all literals from Π as different atoms. $\square$

Comment. This semantics for extended logic programs was proposed by M. Gelfond and V. Lifschitz in their pioneering papers [8, 9], with the following minor difference:

- In many real-life problems, errors are absolutely intolerable, and therefore, we must eliminate all inconsistencies before we can use the knowledge base. In view of this, the original definition from [8, 9] required that we discard stable models that are inconsistent (i.e., that contain both an atom A and its negation $\neg A$), and that if such a stable model is impossible, then we should take the set of *all* literals as an *answer set*. As a result of this approach, even when all the rules and facts that describe some property are, by themselves, quite consistent, if we add unrelated inconsistent rules that describe some other property, we will not be able to deduce anything meaningful about the original property as well.

- In the ever-changing world of software specifications, however, minor changes are very frequent, and minor inconsistencies are inevitable. Of course, it is desirable to eliminate all inconsistencies, but at the intermediate stages, we would like to have a tool that neglects unrelated inconsistencies and provides correct answers for those properties whose description is consistent. Therefore, for our purposes, we use the modified version of the original semantics that allow such "localized" inconsistency.

Closed World Assumption. In some cases, the information describes potentially *open* knowledge: e.g., we have listed some exceptions to the general rule, but additional exceptions may follow.

In many cases, in addition to the knowledge described by the rules (and representable by an extended logic program), we have additional information – that this knowledge is *complete* (*closed*): i.e., that we have enumerated all rules and all exceptions, and if something is not explicitly defined by these rules as an exception, then it is definitely *not* an exception. This additional information, that the knowledge is closed to further changes, is called a *Closed World Assumption* (CWA).

CWA is drastically different from the pieces of information that we have considered before. At first glance, it may therefore seem that formalizing CWA would require a further extension of logic programming. Fortunately, it turns out that CWA can be formalized already in terms of extended logic programming: Namely, CWA states that, for every literal A, if we cannot prove that A is true, then A is false. The condition that we cannot prove that A is true is exactly the condition expressed by negation as failure (*not* A); thus, CWA can be expressed by the following rule

$$\neg A \leftarrow not\ A$$

for every literal A, i.e., in other words, by two rules

$$\neg E \leftarrow not\ E;$$

$$E \leftarrow not\ \neg E;$$

for every atom E.

5.2 How to implement the theoretical concepts of extended logic programming with the existing Prolog compilers

The problem. In the previous section, we have argued that classical negation is important for representing specifications. Therefore, it is desirable that we describe knowledge in terms of *extended* logic programs, i.e., logic program that allow both negation as failure and classical negation.

However, most *practically* available Prolog compilers only allow *generalized* logic programs, i.e., logic programs that may contain negation as failure but not classical negation. In other words, the useful notion of an extended logic program remains largely a *theoretical* notion.

It is, therefore, desirable to implement this theoretical concept of an extended logic program by using a standard logic compiler (that can only handle *generalized* logic programs).

Main idea. The possibility of this implementation follows from the fact that the semantics of an extended logic program were actually defined in terms of the semantics of the auxiliary generalized logic program. Thus, according to this semantic, if we have a logic program Π, and we want to find an answer to the query Q, then we:

- re-formulate Π as a generalized logic program;
- ask *two* queries, Q and $\neg Q$, to this re-formulated program, and then
- combine these two answer into the answer about the original query.

Two problems with this idea. There are two problems with this idea: one minor and one more major.

The minor problem is that most Prolog compilers use only ASCII symbols, so we cannot directly implement the symbol $\neg Q$ for classical negation. This type of a problem is not new for Prolog: the symbol $\leftarrow$ that we use to describe Prolog rules is also not an ASCII symbol, so Prolog compilers usually use a similar-looking combination of ASCII symbols (`:-`). Instead of a non-ASCII expression $\neg Q$ for classical negation, we can use the ASCII combination `neg(Q)`.

The second problem is more disturbing: the above idea seems to require processing *on top* of the Prolog compiler. It is definitely preferable to somehow express this additional processing within Prolog itself. We will show that this is a quite doable.

Solution to the problem. To incorporate the answering mechanism for extended logic programs into Prolog, we propose to introduce a new binary predicate $ans(.,.)$ and add, to the transformed program, the following general rules:

$$\begin{aligned} ans(P, true) &\leftarrow P, not\ \neg P; \\ ans(P, false) &\leftarrow \neg P, not\ P; \\ ans(P, incomplete) &\leftarrow not\ P, not\ \neg P; \\ ans(P, inconsistent) &\leftarrow P, \neg P; \end{aligned} \tag{E}$$

where P stands for an arbitrary atom (we follow a usual Prolog tradition of using capitalized names to describe variables, and names starting with small letters to describe constants). Then, to get an answer for a query Q, we pose the following query to the resulting Prolog program: "$ask(Q, Truth)$?". When faced with a query that has a variable in it, Prolog compilers return the value of the variable that makes this formula true, and a special message (e.g., "false") if there is no such value. We claim that if the underlying Prolog compiler correctly handles generalized logic programs, then for this enlarged program, it will return the correct truth value for every query:

Proposition 1. *Let Π be an extended logic program. Let Π^* be a generalized logic program that is obtained from Π as follows:*

- *First, we interpret all literals from the program Π as new atoms.*
- *Then, we add a new predicate symbol $ans(.,.)$ and new rules* (E) *for every atom P from the original program Π.*

Then, for every atom Q from the original program Π, and for each of the four possible truth values t, $ans(Q, t)$ is true for Π^* if and only if the query Q has a truth value t in the original logic program Π.

Comment 1. For the reader's convenience, we have placed the proofs of this and following propositions into a special appendix at the end of this paper.

Comment 2. If the logic program contains the Closed World Assumption, then the value "incomplete" is impossible, and therefore, we can omit the rule that describes this truth value from (E).

Comment 3. In actual Prolog, the system (E) will take the following form:

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

5.3 Applications to software maintenance

Formulation of the problem. In software maintenance, we want to analyze the relationship between the *original* and the *modified* specifications S and S'. Since, as we have already mentioned, extended logic programming is a natural language for describing these specifications, we can thus assume that we have *two* different extended logic programs Π and Π' that describe, correspondingly, the old and the new specifications. Notice that S and S' correspond to Π and Π' and that the models of Π and Π' correspond to the specified relations $f(S)$ and $f(S')$. We want to investigate, for every query Q, whether the answer to this query has changed when we replace the old specifications with the new ones.

Given system (E), for each query Q, each of the extended programs Π and Π' can give us four different answers, so we have $4 \times 4 = 16$ possible combinations of answers: true-to-true, true-to-false, etc. For example:

- true-to-true means that the statement Q is true in both specifications and is, therefore, consistently *true*;
- true-to-false means that the statement Q was originally true and is now false, i.e., that there is an inconsistency between these two specifications;
- incomplete-to-true means that the new specification *completed* the knowledge expressed in the original specifications, etc.

The simple solution and its drawbacks. As in the last subsection, we can solve this problem if we ask the same query to two different programs and compare the answers. However, it is desirable to have this comparison done by the Prolog program itself.

Solution. We cannot *directly* merge the facts and the rules from the programs Π and Π', because if we did we would lose the information concerning which rules corresponded to the old specifications and which rules corresponded to the new ones. However, we can still merge the logic programs *indirectly*: Namely, in effect, since we have *two* different logic programs, we thus have two different description of each atom P, i.e., in effect, *two* different atoms: "P according to the old specification" and "P according to the new specification". Some atoms are described by the basic facts, so we do not need to duplicate them, but for every other atom P, we can keep this notation for "P in the *original* specification Π", and use a different notation (the natural one is P') to describe this same atom according to the new program Π'.

Since the new program Π' describes the new atoms, we have to replace, in every rule from Π', every atom P that is not a basic fact by the corresponding atom P'. Then, we can merge these programs, add the predicate $ans(.,.)$ to describe the answers to each of them, and add a new predicate to describe comparison:

$$compare(P, T1, T2) \leftarrow ans(P, T1), ans(P', T2). \tag{C}$$

Our claim is similar to the one made in the previous subsection: If the underlying Prolog compiler correctly handles generalized logic programs, then for the thus combined program, will get the correct comparison for every query.

Proposition 2. *Let Π and Π' be extended logic programs with the same alphabet (i.e., with the same atoms), and let Π_0 be a set of facts (called basic) common to both programs. Let Π^* be a generalized logic program that is obtained from Π and Π' as follows:*

- *First, we introduce, for every atom P that does not belong to Π_0, a new atom P', and replace each occurrence of this atom P in the program Π' by the corresponding atom P'.*
- *Second, we combine the rules and facts from the program Π and from the replaced program Π'.*
- *Third, we interpret all literals from the combined program as new atoms.*
- *Fourth, we add a new predicate symbol $ans(.,.)$ and new rules (E) for every atom P that either occurs in the original program Π, or has the form P' for one of such atoms.*
- *Finally, we add the rule (C).*

Then, for every atom Q, and for each pair of possible truth values $\langle t_1, t_2\rangle$, $compare(Q, t_1, t_2)$ is true for Π^* if and only if the query Q has a truth value t_1 in the original logic program Π, and a truth value t_2 in the original logic program Π'.

Comment. In the actual Prolog, we cannot use the notation A' to denote a new atom that corresponds to A; therefore, we will use the ASCII notation `new(A)`. In these new notations, the rule (C) takes the following form:

```
compare(P,T1,T2):-ans(P,T1),ans(new(P),T2).
```

6 An Example

The original specification. Suppose we have a simple specification S to indicate, based upon salary and marital status, when a person is to pay income taxes.

```
pay_taxes([Name,Salary,Marital]):-person(Name,Salary,Marital),Salary > 20000.
```

Furthermore suppose that the complete domain of the problem is:

```
person(bill,19000,married).
person(sam,18000,married).
person(bob,19000,single).
person(joe,18000,married).
person(lyle,21000,married).
person(john,23000,married).
person(sal,34000,single).
person(bart,88000,married).
```

For such a small domain, the specified relation is feasible to compute:

```
pay_taxes([lyle,21000,married]).
pay_taxes([john,23000,married]).
pay_taxes([sal,34000,single]).
pay_taxes([bart,88000,married]).
```

Let us abbreviate the specified relations in the following way:

$f(S) = \{pt(lyle), pt(john), pt(sal), pt(bart)\}$

First modification. Now suppose, a change is made resulting in S':

```
pay_taxes([Name,Salary,Marital]):-person(Name,Salary,Marital),Salary > 15000.
```

The specified relation appears as:
$f(S') = \{pt(bill), pt(sam), pt(bob), pt(joe), pt(lyle), pt(john), pt(sal), pt(bart)\}$
Notice that this change is an example of the leftmost Venn Diagram in Figure 2, where $f(S) \subseteq f(S')$

Second modification. Now suppose the change is made to S, resulting in S'':

```
pay_taxes([Name,Salary,married]):-Salary > 25000.
```

The specified relation appears as:
$f(S'') = \{pt(sal), pt(bart)\}$
In this case, the change is reflected in the middle Venn Diagram of Figure 2, where $f(S) \supseteq f(S'')$.

Third modification. Finally consider the change from S to S''':

```
pay_taxes([Name,Salary,single]):-person(Name,Salary,single), Salary > 15000.
pay_taxes([Name,Salary,married]):-person(Name,Salary,married), Salary > 25000.
```

The specified relation appears as:
$f(S''') = \{pt(sal), pt(bart), pt(bob)\}$
In this case, the third Venn Diagram is represented, where$f(S) \not\subseteq f(S''') \wedge f(S) \not\supseteq f(S''')$.

Applying logic programming techniques to these specifications: general idea. For any problem of reasonable size it is not feasible to compute the specified relations to determine the nature of the change. Therefore, one must hope for a mechanism to detect the need for change "on the fly." The extended logic programming mechanism indeed allows for this form of detection. It turns out that if S is paired with each of the changed specifications S', S'', and S''', the extended logic programming answering mechanism can detect the inconsistencies that result. In other words, when changes to S are combined with S and the extended logic programming answering mechanism is employed, the *comparison* query will result in inconsistent answers for data affected by a change.

Comparing S and S'. For example, when combining S and S':

```
pay_taxes([Name,Salary,Marital]):-person(Name,Salary,Marital),Salary > 20000.
new(pay_taxes([Name,Salary,Marital])):-person(Name,Salary,Marital),Salary > 15000.

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

compare(P,T1,T2):-ans(P,T1),ans(new(P),T2).

neg(P):-not P.
```

Inconsistencies are detected for queries, involving *bob*, *bill*, *sam*, and *joe*. This is because in $f(S)$ these individuals do *not* pay taxes and in $f(S')$ they do pay taxes. For all remaining people, taxes are paid in $f(S)$ and $f(S')$ and the query mechanism answers that it is true that they pay taxes.

Comparing S and S''. When given S and S'':

```
pay_taxes([Name,Salary,Marital]):-person(Name,Salary,Marital),Salary > 20000.
pay_taxes([Name,Salary,married]):-person(Name,Salary,married),Salary > 25000.

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

compare(P,T1,T2):-ans(P,T1),ans(new(P),T2).

neg(P):-not P.
```

Inconsistencies are detected for queries, involving *john* and *lyle*. This is because in $f(S)$ these individuals do pay taxes and in $f(S'')$ they do *not* pay taxes. For all remaining people, taxes are paid or not paid in both $f(S)$ and $f(S'')$ and the query mechanism answers consistently.

Comparing S and S'''. Finally consider S and S''':

```
pay_taxes([Name,Salary,Marital]):-person(Name,Salary,Marital),Salary > 20000.
pay_taxes([Name,Salary,single]):-person(Name,salary,single),Salary > 15000.
pay_taxes([Name,Salary,married]):-person(Name,Salary,married),Salary > 25000.

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

compare(P,T1,T2):-ans(P,T1),ans(new(P),T2).

neg(P):-not P.
```

Inconsistencies are detected for queries, involving *john*, *bob* and *lyle*. For *john* and *lyle*, this is because in $f(S)$ they do pay taxes and in $f(S''')$ they do *not* pay taxes. For *bob*, in $f(S)$ he does NOT pay taxes and in $f(S''')$ he does. For all remaining people, taxes are paid or not paid in both $f(S)$ and $f(S''')$ and the query mechanism answers appropriately.

7 Conclusion

It is our belief that the potential for understanding software evolution and software maintenance is greatly improved through the use of recent results in Logic Programming research. The advantage of this approach is that one can extend Prolog with the simple rules and conventions presented here and actually execute a specification and observe the impact of change.

8 Acknowledgments

Thanks to friends and colleagues, Valdis Berzins, Michael Gelfond, Joseph Goguen, Olga Kosheleva, Vladimir Lifschitz, and Alessandro Provetti 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*, 1990, Vol. 13, pp. 1–18.

[2] V.R. Basili, "Viewing Maintenance as Reuse-Oriented Software Development", *IEEE Software*, 1990, Vol. 7, No. 2, pp. 19–25.

[3] C. Baral and M. Gelfond, "Logic Programming and Knowledge Representation", *Journal of Logic Programming*, 1994, Vol. 19, No. 20, pp. 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*, 1996, Vol. 32, pp. 269–308.

[5] D. Cooke and Luqi, "Logic Programming and Software Maintenance", *Annals of Mathematics and Artificial Intelligence (AMAI)*, A Special Issue on Logic Programming, Nonmonotonic Reasoning, and Action, edited by C. Baral, V. Kreinovich, and V. Lifschitz, 1997 (to appear).

[6] A. M. Davis, "A Comparison of Techniques for the Specification of External System Behavior," *Communications of the ACM*, 1988, Vol. 31, No. 9, pp. 1098–1115.

[7] M. Gelfond and V. Lifschitz, "The Stable Model Semantics for Logic Programming" In: R. Kowalski and K. Bowen, editors, *Proc.* 5^{th} *International Conference and Symposium on Logic Programming*, Seattle, Washington, August 15–19, 1988, pp. 1070–1080.

[8] M. Gelfond and V. Lifschitz, "Logic Programs with Classical Negation", *Proceedings of 7th International Conference on Logic Programming*, Jerusalem, 1990, pp. 579–597.

[9] M. Gelfond and V. Lifschitz, "Classical Negation in Logic Programs and Deductive Databases", *Journal of New Generation Computing*, 1991, Vol. 9, Nos. 3,4, pp. 365–387.

[10] M. Gelfond and H. Przymusninska, "Stratified Extended Logic Programs," draft copy of a paper in preparation.

[11] M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution", *Proceedings of the IEEE*, 1980, Vol. 68, No. 9, pp. 1060–1075.

[12] V. Lifschitz and H. Turner, "Splitting a Logic Program", In: Pascal van Hentenryck (ed.), *Logic Programming. Proceedings of the Eleventh International Conference on Logic Programming*, MIT Press, Cambridge, MA, 1994, pp. 23–37.

[13] 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*, 1995, Vol. 5, No. 1, pp. 89–118.

[14] C.V. Ramamoorthy and D. Cooke, "The Correspondence Between Methods of Artificial Intelligence and the Production and Maintenance of Evolutionary Software", *Proceedings of the Third International IEEE Conference on Tools for Artificial Intelligence*, November, 1991, pp. 114–118.

[15] C.V. Ramamoorthy, D. Cooke, and C. Baral, "Maintaining the Truth of Specifications in Evolutionary Software", *International Journal of Artificial Intelligence Tools*, 1993, Vol. 2, No. 1, pp. 15–31.

[16] J.-P. Tsai and T. Weigert, "A Knowledge-Based Approach for Checking Software Information Using a Non-Monotonic Reasoning System", *Knowledge-Based Systems*, 1990, Vol. 3, No. 3, pp. 131–138.

Appendix A. Proofs

A.1. Proof of Proposition 1

This proof follows from the general result about splittings of logic programs. Proposition 1 follows from the general theory of stable models of *split logic programs*, developed by V. Lifschitz and H. Turner in the paper [12].

In that paper, this notion was defined for class of logic programs that is more general than we have allowed: namely, for the so-called *disjunctive* logic programs that allow an additional connective "or" in the conclusion of "if"–"then" rules, i.e., which allows rules of the type

$$B_1|\ldots|B_d \leftarrow A_1,\ldots,A_n, not\ C_1,\ldots,not\ C_m,$$

where "$B_1|\ldots|B_d$" means "B_1 or ... or B_d". Since in Proposition 1, we only consider logic programs without disjunction, a reader who is not interested in this general result can simply assume that we have a regular logic program. (We will return to the general case in Appendix B.)

In Appendix B, we will propose a generalization of the result from [12]. To make this generalization easier to describe, and also to simplify the result's understanding by the reader who may not be well trained in logic programming, we will slightly modify the original notations and expositions (while preserving the construction from [12] intact).

Splitting: main definitions. The main definition from [12] can be reformulated as follows: for every two literals a and b from a logic program, let us denote $a \geq b$ if in one of the rules, a appears in the head (i.e., in the conclusion part of the rule), and b appears elsewhere in this rule, i.e., either in its head (as one of the possible rule's conclusions) or in its body (as one of the conditions of the rule). By a *splitting* of the program, we mean a mapping s from the set of all literals into a linearly well-ordered set (i.e., into the set of all ordinal numbers that are smaller than some ordinal number μ) for which $a \geq b$ implies $s(a) \geq s(b)$. In other words, to every literal a, we assign a *level* $s(a)$.

Comment. For Propositions 1 and 2, we do not need *infinite* ordinal numbers; since finite ordinal numbers are exactly natural numbers $0, 1, 2, 3, \ldots$, a reader who does not feel comfortable with general ordinal numbers can use natural numbers instead. In this case, instead of a *transfinite* recursion, i.e., recursion over ordinal numbers, the reader can substitute normal recursion, i.e., recursion over natural numbers.

Answer sets for split logic programs: intuitive description. Intuitively, the existence of a splitting sequence means that rules that define literals from level α only use literals from this and lower levels. Thus, e.g., rules that define literals of level 0 only use literals from the same level and thus, these rules are self-sufficient to define which of the literals of level 0 are true and which are not.

As soon as we have defined the truth values of all literals of level 0, we can define the truth values of literals of level 1, etc.

Answer sets for split logic programs: a formal description. Formally, this process corresponds to *transfinite recursion*, i.e., recursion over all possible levels (in our case, over all ordinal numbers $< \mu$; if μ is finite, this becomes a simple recursion):

- First, we take all the rules whose heads are of level 0. By definition of a splitting, all conditions from these rules are also of level 0. Then, we find an answer set A_0 for the the corresponding logic program.
- If for some level α, we have already described the answer sets A_ν for all levels $\nu < \alpha$, then we can define the answer set A_α corresponding to this level α as follows:

- consider the union $A_{<\alpha}$ of all already defined sets A_ν;
- select all the rules whose heads contain only literals of levels α and lower;
- delete all the rules in which one of conclusions in the head is a literal from the set $A_{<\alpha}$ (because these rules are automatically true);
- if one of the conclusions of a rule is a literal of level $< \alpha$ that does not belong to $A_{<\alpha}$, we delete this literal from the conclusion (because this literal cannot be true);
- delete all the rules in which one of the conditions is *not* p for some literal p included in $A_{<\alpha}$ (intuitively, since $p \in A_{<\alpha}$, the condition p is true and thus, the opposite condition *not* p is not satisfied);
- delete all the rules in which one of the conditions is p for some literal p of level $< \alpha$ that is not included in $A_{<\alpha}$ (intuitively, since $p \notin A_{<\alpha}$, the condition p is not true and thus, the rule is not applicable);
- from each of the remaining rules, delete all conditions p and *not* p that are literals of levels $< \alpha$ (after our previous deletions, all these conditions are automatically true);

As a result, we get a logic program which only contains literals of level α. If this logic program has an answer set, we add all literals from this answer set to $A_{<\alpha}$ and get A_α. (If this logic program turns out to be empty, we take simply $A_{<\alpha}$ as A_α.)

As a result of this procedure, we get a set A_μ.

The main result about split logic programs. The main theorem from [12] consists of the following two statements:

- if this set A_μ is consistent, then it is an answer set of the original logic program;
- vice versa, every consistent answer set A to the original logic program can be obtained by this transfinite recursive procedure.

For *stable models* (and, correspondingly, for s-answer sets), similar results are true without requiring consistency.

Proof of Proposition 1. To apply this general result to our program Π^*, we can take a splitting into two levels: at the top level, we have all literals $ans(.,.)$, and at the bottom level, all other literals. Since we only have two levels, we only have to consider A_0 and A_1. Here, Π consists exactly of all the rules whose heads are of level 0; when we have found an s-answer set for this program, then, according to the recursive procedure, for every rules with the conclusion $ans(.,.)$, either the rule itself will be eliminated, or all its conditions will be eliminated. It is easy to check that as a result, we get exactly the literals $ans(.,.)$ that we want. Proposition 1 is proven.

A.2. Proof of Proposition 2

Proposition 2 can be proven by a similar use of a theorem about split logic programs; the main difference is that, in contrast to the proof of Proposition 1, where *two* layers were sufficient, here, we need *five* different layers. The corresponding splitting map $s(p)$ is as follows:

- to all literals p from the set Π_0, we assign the lowest level ($s(p) = 0$);
- to all literals from the program Π, we assign level 1;
- to all literals from the program Π', we assign level 2;
- to all literals of the type $ans(P,.)$, we assign level 3;
- finally, to all literals of the type $compare(.,.,.)$, we assign level 4.

Then:

- On level 0, we will have all facts from Π_0 true.
- On level 1, we will have an answer set for Π.
- On level 2, we will have an answer set for Π'.
- On level 3, we will have the correct answers for $ans(.,.)$,
- Finally, on level 4, we will have correct answers for $compare(.,.,.)$.

Comment. This proof is technically correct, but somewhat un-natural. In the next appendix, we will show how the theorem about split logic programs can be generalized in such a way that results like Proposition 2 become more naturally provable.

Appendix B. A New Result about Splittings of Logic Programs

B.1. Motivations

In the proof of Proposition 2, there was a definite logic in the relative levels from Π_0, Π, $ans(.,.)$, and $compare(.,.,.)$, but there was no particular logical reason why literals from Π' were assigned a higher level than literals from Π: it could be done the other way around. There is no *logical* relation between Π and Π', and the only reason why we placed one above another was *technical*: we had to somehow order them simply because the theorem about split logic programs (that we were using) requires that the levels are *linearly* ordered. Natural logical relations between the different different parts of the program Π^* lead only to a *partial* order between different parts:

$$\begin{tikzcd}
 & compare(.,.,.) \arrow[d] & \\
 & ans(.,.) \arrow[dl] \arrow[dr] & \\
\Pi \arrow[dr] & & \Pi' \arrow[dl] \\
 & \Pi_0 &
\end{tikzcd}$$

It is, therefore, desirable to *generalize* Lifschitz's and Turner's result to logic programs for which the splitting map maps literals into a *partially* ordered set.

B.2. Formulation of the new result

Fortunately, this generalization can be obtained by a simple modification of the original proof. As in the proof of Proposition 1, for every two literals a and b from a logic program, we use the denotation $a \geq b$ to indicate that in one of the rules, a appears in the head (i.e., in the conclusion part of the rule), and b in the body of this rule.

Let us recall that a (partially) ordered set M is called *well-ordered* if it does not have an infinite monotonically decreasing sequence $m_1 > m_2 > \ldots > m_n > \ldots$

Def. By a *generalized splitting* of a logic program, we mean a mapping s from the set of all literals into a well-ordered set (not necessarily linearly ordered) for which $a \geq b$ implies $s(a) \geq s(b)$. $\square$

For a program that allows a generalized splitting, we can, almost literally, repeat the construction described in the proof of Proposition 1, and get the following result:

Theorem. *If a logic program allows a generalized splitting, then:*

- *if a set A_μ obtained by the above-described transfinite recursion is consistent, then it is an answer set of the original logic program;*
- *vice versa, every consistent answer set A to the original logic program can be obtained by the above transfinite recursive procedure.*

B.3. Proof of the new result: main idea

There are two main possibilities to prove this result:

- One possibility is to simply repeat the proof from [12].
- Another possibility is to take into consideration the fact that every well-ordering can be extended to a *linear* well ordering and therefore, we can apply the original theorem from [12] to prove our result. From the recursive construction, it easily follows that if in the original ordering, levels α and β were unrelated by the ordering relation, then the corresponding reduced logic programs on stages α and β do not depend on each other (similarly to levels corresponding to Π and Π' in the proof of Proposition 2).