How To Combine Nonmonotonic Logic and Rapid Prototyping To Help Maintain Software*

Luqi
Computer Science, Naval Postgraduate School, Monterey, CA 93943

Daniel Cooke
Computer Science, University of Texas at El Paso, El Paso, TX 79968

November 2, 1995

Abstract

This paper explores the possibility of automated support for detecting inconsistencies in software systems and requirements. The inconsistencies are introduced when the environment of the software system changes. We refer to the software environment as its context. We review the recent research progress on nonmonotonic logics, pointing out the significance of these results to software maintenance. We explain how a practical implementation of such logics can be obtained via a simple extension to logic programming in the form of an answer procedure that realizes the Extended Logic Semantics [7] for nonmonotonic logic programs that have a unique answer set (which is a large and useful class of logic programs).

We augment the existing automated capabilities of the Computer Aided Prototyping System (CAPS) for rapid prototyping via the extension to logic programming to provide an improved automated capability for detecting certain kinds of inconsistencies created by implicit requirements changes. We illustrate the significance of this capability via an example prototype for a problem originally suggested by Lehman.

1 Introduction

1.1 Costs of Software Evolution

Software evolution accounts for more than half of the total software cost. Each time a new software system is put into use, some fraction of the work force must be devoted to its *maintenance*. If it is assumed that all systems require some maintenance effort and a constant work force engages in software development for a long time, the fraction of effort available for developing new systems will get small, and can be kept from vanishing only by retiring or replacing some old systems [16]. There has been a great deal of interest in reducing software evolution costs. The reduction of evolution costs may involve some level of automated support for maintenance. This paper explores the extent to which it is possible to automatically detect the need for maintenance.

It is well known that 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.

*Research sponsored in part by the NSF under grant numbers CCR-9058453 and CDA-9015006; by the AFOSR under contract F49620-93-1-0152; by the ARO under grant number ARO-145-91, and by NASA Ames Research Center NAG 2-670 Supp. #2.

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

Figure 1: Context Changes Trigger Software Changes.

Prototyping is one promising approach to reduce the amount of corrective maintenance [6]. 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 represents responses to requirement changes and reflect the kind of change that occurs naturally in the software environment (or context). Even if software is valid and verified, adaptive changes continue to be necessary. In this paper, we focus strictly on the detection of the need for adaptive maintenance.

1.2 Adaptive Changes and Software Maintenance

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, the 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 case, 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 is 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 [9] 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: $C_i \downarrow S_i$ and $S_i \downarrow P_i$. A context shift should result in maintenance activity where valid changes are made to a specification and correct changes are made to a program.

Notice that each P_i (where $i > 0$) is actually a completely new system compared to P_{i-1}. In general, the main difference between the production of P_0 and a later version P_i is the fact that in the production of P_i a great deal of software from P_{i-1} is reused. The maintenance process suggested here is identical to Basili's Full Reuse Maintenance Process [2].

Software evolution should be viewed as a continual process of re-validation and re-verification. The contribution made in this paper is the use of a logic to detect the need for maintenance. To detect context shifts, we need to detect when the current software is based upon a specification that is no longer valid. In other words, a context shift results in an invalid specification - the specification that correctly reflects C_i does not correctly reflect C_{i+1}.

In order to show the feasibility of our approach, we combine results from the area of nonmonotonic logics to a Prolog version of the Computer Aided Prototyping System (CAPS). We choose CAPS because it provides a well defined, high level language and is easy, as a result, to combine with the nonmonotonic logic semantics. We believe that it is possible to extend the results we have obtained with CAPS to an operational software environment.

1.3 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.

This discussion of immutable and mutable specifications is analogous to Lehman's S- and E-type programs. The types are based upon the nature 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 and 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. We focus on E-type and S-type programs.

An E-type program has to evolve because the validity of the assumptions coded into the program change with time. Lehman [10] gives a dramatic example involving British ships in the Falkland Island War. The software system defending the ships was based, in part, on the assumption that an EXOCET missile is friendly. The assumption had been true until the Falkland Island War in which a British ship was sunk by an EXOCET. The assumption in an E-type program is a mutable specification. It is the mutable specifications (or assumptions) which serve as the seed for 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 [18, 19, 21]. Nonmonotonic logics provide formalisms to handle beliefs (or assumptions). Intuitively, nonmonotonic logics allow the retraction of beliefs when new information is presented which contradicts those beliefs. 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. A specification is viewed here, as a set of assertions (both extensional and intensional) in predicate logic. 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'. If the specification is valid, the specified relation relates each input of a program to all output values that can be valid responses to the input according to the software context. 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 $\forall S, S' (S \subseteq S' \Rightarrow f(S) \subseteq f(S'))$. (Where $\Rightarrow$ denotes implication.) $\Box$

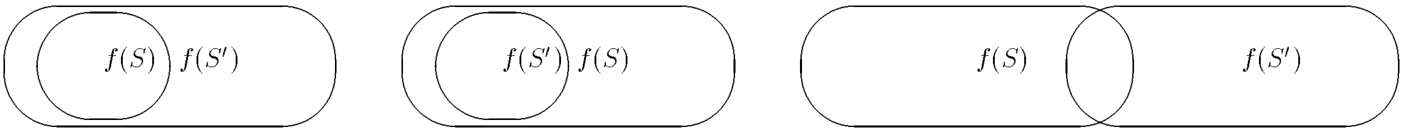
The definition of nonmonotonic essentially negates the formula above:

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

The definition of nonmonotonic suggests a classification for 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. $f(S) \supseteq f(S')$;



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

Figure 2: Relationships among specifications due to change.

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'. *A change to the specification $S \neq S'$ is detectable through the detection of an inconsistency:*

1. $f(S) \subseteq f(S')$ means $\exists p(p \notin f(S) \wedge p \in f(S'))$ where p is an input/output pair;
2. $f(S) \supseteq f(S')$ 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 automatic 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.

Our goal is to demonstrate that the results from the study of nonmonotonic logics provide the necessary framework for detecting the need for adaptive maintenance. We then apply these results to a practical system. Section 2 explores different forms of mutable specifications based on results from the study of nonmonotonic logic. In section 2 the framework for detecting the need for adaptive maintenance (i.e., the detection of context shifts) is presented. Section 3 applies the framework to the CAPS model and demonstrates that the integrated model possesses the facility to detect the context shift and the need for maintenance.

2 Overview of Context Dependent Specifications

2.1 Introduction

There are three different kinds of *mutable specifications*:

1. Specifications whose validity depends on time - i.e., valid in some possible contexts and invalid in others;

2. A distinguished subset of 1: specifications which are usually valid but there exist exceptions - e.g., in general birds can fly, however penguins are an exception; and

3. An incomplete specification (missing specifications). Such a specification brings with it an implicit assumption that the cases that are left undefined will not occur in practice.

We illustrate these three types of mutable specifications with three logic programs. The first is a situational logic program in which some of the context dependent concerns are handled (type 1 above). The second program is based upon autoepistemic logic. It handles conflicts with default assumptions as well as problems with reasoning with incomplete information (types 1, 2 and 3 above). The last program extends the autoepistemic logic-based program to detect context shifts in which specifications that once were valid are later found to be invalid (covering all three specification types).

All of the examples in this paper have executed using Quintus Prolog Release 3.1.1 for the SUN SPARCstation 2.

2.2 Situational Logic

Situational logic adds an extra parameter called a situation to every predicate to explicitly represent the dependence of the truth of the predicate on the context or state of the world at some given time. Consider a standard situational logic program to determine if a student is to be considered an honors student. For the purposes of this example, an honors student never makes a grade below 90.

PROGRAM 1.

```
  /* Model of the application. */
  /* The constant s0 represents the initial situation. */
  /* Situation S becomes situation grade(G, S)
     after making the grade G. */
holds(honors(X), S) :- high_grades(X, S).

high_grades(X, grade(G, s0)) :- is_high(G).
high_grades(X, grade(G, S)) :- is_high(G), high_grades(X, S).
  /* high_grades is not defined in so. */
```

```
is_high(G) :- number(G), 90 =< G, G =< 100.

  /* Reusable system code: front end for situational logic */
ans(P, true) :- P.
ans(P, false).
```

This program uses the standard Prolog answering mechanism (represented by the `ans` predicate) for an arbitrary goal `P`:

- if `P` can be proven then the answer is true,
- otherwise the answer is false.

We expect all queries to be expressed via the answer predicate to enable delivery of both positive and negative Boolean results. The *ans* predicate will gain more significance in the next two versions of the program, where it will be extended to incorporate features of nonmonotonic logic.

The program represents the state of the student's progress in the situation variable S and effectively checks if the property honors holds for student X in state S. Consider the following queries and the resulting answers:

```
(a) ?- ans(holds(honors(bob), s0), A).
       A = false
(b) ?- ans(holds(honors(joe), grade(91, grade(95, s0))), A).
       A = true
(c) ?- ans(holds(honors(sam), grade(89, grade(95, s0))), A).
       A = false
(d) ?- ans(holds(honors(art), grade(98, grade(80, s0))), A).
       A = false
(e) ?- ans(holds(honors(bill), grade(incomplete, grade(95, s0))), A).
       A = false
(f) ?- ans(holds(honors(dirk), grade(incomplete, grade(89, s0))), A).
       A = false
```

The answers to the queries (b), (c), (d), and (f) are valid (i.e. correct relative to our expectations and experience). However, this version of the program has two problems: in query (a) the answer false is obtained when in fact there is no data on which to base an answer; and in queries (e) and (f) null values have been entered for some grades, making the false answer incorrect for (e) and correct by accident for query (f). Even though the information concerning Dirk is incomplete, the answer to query (f) is correct because there is enough information to know that he cannot be an honors student.

The problematic answers are due to the fact that the context has presented data which is exceptional. Although the program accepts this exceptional data, it is giving answers which are incorrect. The default assumptions implicit in the initial model of the problem domain include: (1) any student considered for honors will have taken a class (with which (a) conflicts) and (2) any grade obtained will be a number in the range 0 through 100 inclusive (with which (e) and (f) conflict).

Note that one aspect of nonmonotonicity is captured in program 1. The answer to (b) is true for the situation given, but the answer to the same query relative to a possible future situation is false:

```
(g) ?- ans(holds(honors(joe), grade(75, grade(91, grade(95, s0)))), A).
       A = false
```

Although Joe was considered to be an honors student after receiving a 91 and 95, his most recent grade (i.e., a 75) disqualifies him as an honors student. This example and the previous query (b) demonstrate the temporal aspects of nonmonotonic reasoning in situational logic (i.e., nonmonotonicity includes the situation where information that becomes available at a later point may falsify earlier answers).

2.3 Handling Conflicts With Default Assumptions

Several formalisms have been developed for nonmonotonic reasoning which provide varying degrees of applicability to software context monitoring. Among these are Circumscription [14], Nonmonotonic modal logics [4, 15], Autoepistemic Logic [17], etc. These works have been surveyed in the context of the concern for mutable specifications in [18, 19].

This section presents a standard mechanism for handling exceptions to default assumptions based on autoepistemic logic, using a front-end answering mechanism to Prolog. The front-end presented was developed to study the "Yale Shooting Problem". The autoepistemic logic front-end provides the ability to:

1. answer yes, no, or unknown;

2. reason using true negation (i.e., assertions of facts known to be false);

3. support exceptions to general rules; and

4. reason with null values [20] or account for incomplete knowledge (i.e., the closed world assumption).

The ability to process an equivalence (i.e., iff) is also an aspect of support for autoepistemic logic, but the implications of this feature are beyond the scope of this paper. A restricted form of equivalence is implementable in Prolog.

To handle incomplete specifications, the series of answer clauses of the autoepistemic logic front-end provides the ability to answer yes, no, or unknown:

```
ans(P, true) :- P.                              (1)
ans(P, false) :- not(P).                        (2)
ans(P, unknown).                                (3)

not(not(P)) :- P.                               (4)
```

As before, all queries are routed through the answer predicate. The answer to a query `P` is true if `P` is proven in (1); the answer is false if `not(P)` is proven in (2); and otherwise, the answer is unknown in (3). Since `not` is not a Prolog primitive, we also have to explicitly tell the system in (4) that `not(not(P))` means the same thing as `P` to make the answer mechanism work properly when `P` has the form `not(Q)`.

The concept of state is essential in autoepistemic logic to allow the truth of a query to vary depending on the context. For example, if one shoots a gun that has not been previously loaded, it is false that a bullet will discharge. Autoepistemic logic achieves this by using situations in the same way as situational calculus. Autoepistemic logic also provides classical negation through explicit assertion of the conditions which make a predicate false, as illustrated by the following examples.

```
not(P, S).                                           (5)
not(P, S) :- q_1, ..., q_n.                          (6)
```

In (5), `P` is asserted to be false in the situation `S`. In (6), `P` is false in situation `S` if `q_1` through `q_n` are proven. Note that any `q_i` may itself be negated.

Another feature of autoepistemic logic is its ability to express default assumptions (general rules that may have some exceptions). This is accomplished by asserting abnormalities for general rules as follows:

```
abnormal(P, S).                                      (7)
```

In (7), the specifier is stating that the situation `S` is abnormal with respect to the literal `P`. Default assumptions are expressed as statements about all situations that are not known to be abnormal. For example, a student is normally an honors student as long as the grades made thus far (i.e., in the given situation) are satisfactory. A student is abnormal with respect to being an honors student when a situation is reached wherein some other event has occurred that could disqualify the student from becoming an honors student. Incompleteness of knowledge is captured via the negation as failure rule (NFR) as follows.

```
not_known(P) :- call(P), !, fail.                    (8)
not_known(P).                                        (9)
```

The literal `not_known(P)` means that one does not have reason to believe that `P` is true. Therefore, if `P` can be proven, then `not_known(P)` fails (8). Otherwise, `not_known(P)` is true (9).

Program 2 and the following examples illustrate how features of autoepistemic logic can be used to address the problems of queries (a), (e), and (f) in Section 2.2.

PROGRAM 2.

```
    /* Model of the application. */
    /* The constant s0 represents the initial situation. */
    /* Situation S becomes situation grade(G, S)
       after making the grade G. */
|   /* Situation S becomes situation retake(G, S)
|      after retaking a course with grade G. */
  holds(honors(X), S) :- high_grades(X, S),
|                        not_known(abnormal(honors(X), S)).
| not(holds(honors(X), S)) :- not(high_grades(X, S)).
```

```
  high_grades(X, grade(G, s0)) :- is_high(G).
  high_grades(X, grade(G, S)) :- is_high(G), high_grades(X, S).
    /* high_grades is not defined in so. */
| high_grades(X, S) :- functor(S, F, N), F \== grade, arg(N, S, Last_S),
|                      high_grades(X, Last_S).
|   /* This invariance rule says that events other than getting a grade
|      do not affect the high_grades predicate. */

| not(high_grades(X, grade(G, S))) :- not(is_high(G)).
| not(high_grades(X, S)) :- functor(S, F, N), arg(N, S, Last_S),
|                           not(high_grades(X, Last_S)).
|   /* This invariance rule says that if the high_grades predicate is
|      false it will remain false in all possible future situations. */

  is_high(G) :- number(G), 90 =< G, G =< 100.
| not(is_high(G)) :- number(G), 0 =< G, G =< 90.

| abnormal(honors(X), S) :- repeated_course(S).

| repeated_course(retake(G, S)).
| repeated_course(S) :- functor(S, F, N), arg(N, S, Last_S),
|                       repeated_course(Last_S).
|   /* Subsequent events do not affect whether or not a course
|      has been repeated. */

   /* Reusable system code: front end for autoepistemic logic */

  ans(P, true) :- P.
| ans(P, false) :- not(P).
| ans(P, unknown).

| not(not(P)) :- P.

| not_known(P) :- call(P), !, fail.
| not_known(P).
```

The parts of program 2 that differ from program 1 are highlighted using vertical bars (|) in the left margin (the bars must be removed to execute the program). Program 2 extends the model of the application to include another kind of event that can affect the honors status of a student - retaking a course. The program also includes invariance rules that enable it to ignore other kinds of events (i.e. other situation-constructing functions recording events that do not affect the honors status of a student). This removes the assumption that there is only one kind of event, which is implicit in program 1. We have thus transformed the model into a form that can be consistently combined with models of other aspects of the application, such as whether or not a student is eligible for financial aid. This is significant because models for real (i.e., complicated) systems are understandable only if they can be factored into independent parts that can be analyzed in isolation.

We have also refined the model by adding explicit definitions of the default assumption that normal honors students do not retake courses, and by adding rules for deriving negative assertions. To see the effects of these changes, reconsider the queries (a) - (f):

```
(a') ?- ans(holds(honors(bob), s0), A).
        A = unknown
(b') ?- ans(holds(honors(joe), grade(91, grade(95, s0))), A).
        A = true
(c') ?- ans(holds(honors(sam), grade(89, grade(95, s0))), A).
        A = false
(d') ?- ans(holds(honors(art), grade(98, grade(80, s0))), A).
        A = false
(e') ?- ans(holds(honors(bill), grade(incomplete, grade(95, s0))), A).
        A = unknown
(f') ?- ans(holds(honors(dirk), grade(incomplete, grade(89, s0))), A).
        A = false
```

Note that queries (a'), (e'), and (f') now give valid answers. These answers are obtained due to the improvement in the answering mechanism (`ans`), the treatment of null values (e.g., "incomplete" in (e') and (f')) in the `is_high` procedure, and reasoning with negated information. These are some of the standard features used by the autoepistemic logic improvement to handle conflicts with default assumptions.

For the situational logic answer procedure, if it was not possible to prove a predicate, the predicate was assumed to be false (this is Negation as Failure - NFR), leading to an incorrect result for query (a). The autoepistemic answer procedure, on the other hand, will produce "false" only if it can prove that the predicate is false. When the autoepistemic logic answer procedure is applied to query (a'), it is not possible to prove or disprove the goal `holds(honors(bob), s0)`. Therefore, the correct answer "unknown" is obtained. This handles the exception to the default assumption that any student considered for honors will have at least one grade.

The autoepistemic logic answer mechanism supports reasoning about null values by explicit assertion of negative information, as illustrated by queries (e') and (f'). Situations of the form `grade(incomplete, S)` are interpreted to mean that a student has a grade of "incomplete" or "in-progress" for some class. The definition of the `is_high` predicate has been extended to explicitly define when the predicate is false as well as when it is true, thus making the expected range of values for normal grades apparent. This handles null values in the decision making as follows.

In query (e'), `honors(bill)` cannot be shown to be true because the constant "incomplete" cannot be proven to be a high grade because it is not a number. Similarly, "incomplete" cannot be proven not to be a high grade, so that the negation of honors cannot be proven either. Consequently, the answering mechanism produces the answer "unknown". Thus, the default assumption that all grades are numbers in the range from 0 to 100 is represented explicitly in the positive and negative assertions about the `is_high` predicate, and is handled properly in the reasoning process. In contrast, query (f') contains enough information to determine that Dirk is definitely not an honors student, regardless of the null value and the fact that the truth value of `is_high(incomplete)` is not known, because the appearance of the grade 89 in the situation of query (f') enables the system to prove that `high_grades` is false.

Another kind of exception to a general specification is illustrated by the following query:

```
(h) ?- ans(holds(honors(john), grade(98, retake(91, grade(99, s0)))), A).
       A = unknown
```

In query (h), John would normally be considered an honors student based on the grades he has made. However, the fact that he has retaken a class makes him exceptional: a student who has retaken a class is not judged to be an honors student according to the same criteria as a student who has made the same grades without retaking a course. Note that the abnormality condition says only that the rules about determining honors from **high_grades** do not apply to students with repeated courses, and it does not disqualify such students from the honors category. It does say that additional criteria are needed for determining whether or not a student with repeated courses is an honors student, as reflected by the answer to query (h). Rules covering this case can be added to the specification without affecting the completely defined part of program 2 (i.e. the results of all queries that currently produce "true" or "false" will be unaffected, but queries that produce "unknown" may be refined to produce more definite answers).

The last example illustrates the effect of declared abnormalities: they are retractions that make some previously defined answers become undefined. This prepares the way for future conservative extensions to the theory defined by the specification, and supports the application of monotonic transformations to organize the history of an exploratory development [5]. Such a view is useful because several different extensions are often explored in a development effort, particularly in the context of prototyping. Each extension corresponds to a different alternative in the design of the functional specification for the proposed system. Information elicited from the users via experiments with the corresponding versions of the prototype is used to evaluate the alternatives and to choose one of them. This process is enhanced if the concrete representation of the specification can be separated into a common part and several different extension parts, so that after one of the alternatives is adopted it is easy to determine which part of the specification to keep and which parts to remove and archive as explored but inferior alternatives. The definition of the abnormalities associated with a concept, as illustrated in program 2, is a convenient mechanism to pin down the boundaries of the common (or certain) part of a specification in preparation for an exploration of several competing formulations of an unresolved issue. The effect of the abnormality is to leave a neat hole in the domain of the specification, which can be filled with several different refinements corresponding to the competing solutions for resolving the issue in question.

2.4 Detecting Context Shifts

Recall figure 1. It is a context shift that requires adaptive maintenance. The parts of the context which may change, resulting in a context shift, are reflected by mutable specifications. The desire here is to detect any possible context shift. The formulae in section 1.3. make it clear that one must be capable of detecting

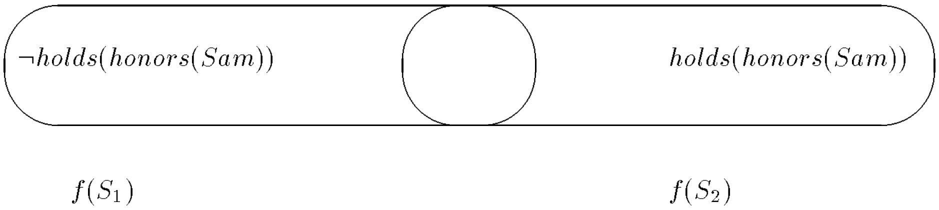


Figure 3: The Intended Relationship.

inconsistencies between the model of the software context and the actual context. In this section, we elaborate how one detects the context shift by a further extension to the Prolog front-end presented in program 2.

It is apparent that program 2 can deal with many of mutable changes that may appear in the problem domain. Suppose, however, that the following information becomes known to the program (i.e. is asserted in the database):

```
holds(honors(sam), S).
```

Consider the following query:

```
(i) ?- ans(holds(honors(sam), grade(98, grade(80, s0))), A).
```

If the new information about Sam is asserted at the beginning of the autoepistemic logic program (i.e., program 2), the answer to this query is “true” under program 2. However, there is a problem. The normal basis for determining honors has been overridden: Sam has been asserted to be an honors student even in situations such as the one in query (i), where Sam has made an 80 (i.e., a grade less than 90). This situation indicates that a context shift from the initial context C_0 to a new context C_1 has taken place.

If we look more closely at the problem, we find that there is an undetected inconsistency in this example. It is clear that $S_1 \subseteq S_2$: some information has been added to the specification, and none has been removed. Specifically, $holds(honors(Sam), X) \notin S_1 \wedge holds(honors(Sam), X) \in S_2$. Furthermore, the intent of the change is $f(S_1) \not\subseteq f(S_2)$. See figure 3.

Since the program contains both $f(S_1)$ and $f(S_2)$, it is possible to prove that Sam is an honors student, and it is also possible to prove that Sam is not an honors student. A minor extension to the autoepistemic logic answer mechanism will, in general, enable the detection of context shifts when exploratory queries result in such inconsistencies, as illustrated below.

PROGRAM 3.

```
    /* Model of the application. */
    /* The constant s0 represents the initial situation. */
    /* Situation S becomes situation grade(G, S)
       after making the grade G. */
    /* Situation S becomes situation retake(G, S)
       after retaking a course with grade G. */
  holds(honors(X), S) :- high_grades(X, S),
                         not_known(abnormal(honors(X), S)).
  not(holds(honors(X), S)) :- not(high_grades(X, S)).

| holds(honors(sam), S).
```

```
| not(holds(honors(joe), S)).

  high_grades(X, grade(G, s0)) :- is_high(G).
  high_grades(X, grade(G, S)) :- is_high(G), high_grades(X, S).
    /* high_grades is not defined in so. */
  high_grades(X, S) :- functor(S, F, N), F \== grade, arg(N, S, Last_S),
                       high_grades(X, Last_S).
    /* This invariance rule says that events other than getting a grade
       do not affect the high_grades predicate. */

  not(high_grades(X, grade(G, S))) :- not(is_high(G)).
  not(high_grades(X, S)) :- functor(S, F, N), arg(N, S, Last_S),
                            not(high_grades(X, Last_S)).
    /* This invariance rule says that if the high_grades predicate is
       false it will remain false in all possible future situations. */

  is_high(G) :- number(G), 90 =< G, G =< 100.
  not(is_high(G)) :- number(G), 0 =< G, G =< 90.

  abnormal(honors(X), S) :- repeated_course(S).

  repeated_course(retake(G, S)).
  repeated_course(S) :- functor(S, F, N), arg(N, S, Last_S),
                        repeated_course(Last_S).
    /* Subsequent events do not affect whether or not a course
       has been repeated. */

   /* Reusable system code: front end for stable model semantics */

| ans(P, inconsistency) :- P, not(P).
  ans(P, true) :- P.
  ans(P, false) :- not(P).
  ans(P, unknown).

  not(not(P)) :- P.

  not_known(P) :- call(P), !, fail.
  not_known(P).
```

As before, differences from the previous version are highlighted using vertical bars on the left margin. The final version of the answer mechanism shown in program 3 is suggested by Table 1. Table 1 presents the series of answering mechanisms (*CWA* for Closed World Assumption; *AE* for Autoepistemic Logic: and *EL* for Extended Logic) that have arisen out of the study of logic programming and nonmonotonic logics. The most powerful mechanism is *EL* which is based upon the Extended Logic Semantics [3]. Table *EL* shows us that there are four possible outcomes if we try to determine the truth value of an arbitrary predicate `p` by attempting to construct proofs. In Table 1, a hyphen, '-', indicates there is no attempt to prove the goal which labels the associated column. For example, in the *CWA* subtable, if the attempt to prove `p` fails, then the answer is $false$.

If, in Extended Logic Programming, the attempt to prove `p` succeeds and the attempt to prove `not(p)` fails, then we can safely conclude that `p` is true. Similarly, when `not(p)` can be proved and `p` cannot be proved then we can safely conclude that `p` is false. However, it is possible that we succeed in proving both `p` and `not(p)`

CWA			*AE*			*EL*		
p	not(p)	Answer	p	not(p)	Answer	p	not(p)	Answer
success	-	true	success	-	true	success	failure	true
failure	-	false	failure	success	false	failure	success	false
			failure	failure	unknown	failure	failure	unknown
						success	success	inconsistent

Table 1: Possible Outcomes for Query p.

(i.e., an inconsistency). In such a case, the program possesses an inconsistency which is not detectable by the standard Prolog answering procedure or even by the autoepistemic answering procedure, because the conflict is masked by the answer given by the first clause to succeed. Finally, it may not be possible to prove either `p` or `not(p)`, which means that the program does not possess sufficient knowledge to give an answer.

Notice that programs 2 and 3 have two forms of negation. The `not_known` is an implentation of the negation as failure rule (NFR). The NFR implements the closed world assumption which, given an arbitrary predicate, `p`, states that `p` is not true if it is not possible to prove `p`.

The other form of negation in the programs, `not`, is classical negation. In other words, if it is known that `p` is false, `not(p)` is explicitly stated in the logic program.

Our answering mechanism (not to be confused with the answer set) extends Prolog to implement a restricted version of the Extended Logic Semantics. The extension implements the Extended Logic Semantics for logic programs with unique *answer sets* by considering all of the *answering* possibilities as shown in Table 1. The class of programs with unique answer sets is large [8], and we believe that it contains all specifications that are sufficiently complete to be considered for the purposes of a software development project.

An answer set is a deductive closure of a set of rules where NFR (negation as failure) is treated in accordance with the closed world assumption. (See [3] for details on the construction of answer sets.) An incomplete specification without negation as failure corresponds to multiple models but only one answer set that contains the features common to all models. Let A be the answer set of a program *P1*. If g is a goal of the logic program, then g is true if $g \in A$. Notice that NFR has a great deal of significance in the discussion of answer sets. NFR succeeds for goal g when $g \notin A$. If a logic program, *P2*, contains:

```
p :- not_known(q).
q :- not_known(p).
```

then multiple answer sets are obtained because p and q are answers based upon mutually exclusive conditions relative to the answer sets. Specifically, $p \in A_1$ *if* $q \notin A_1$ and $q \in A_2$ *if* $p \notin A_2$. Thus, multiple answer sets occur due to mutually exclusive conditions on membership in the answer set(s) of a program.

One can clearly see that a program that does not contain NFR is guaranteed to have a unique answer set. Furthermore, a program may contain an NFR that is not contained in a loop (i.e., there are no *negative loops* in the program). A program without negative cycles is guaranteed to have a unique answer set. Since *stratifiable programs* [1] do not contain negative cycles, stratifiable programs have unique answer sets [3]. Program 3 is a stratifiable program and, thus, has a unique answer set.

The answer mechanism presented in program 3 computes the intersection of the answer sets of a logic program. For programs with unique answer sets, the answer mechanism implements the Extended Logic Semantics. For programs with more than one answer set, the answer mechanism will provide the correct answer for queries which are contained in the intersection of the answer sets. For queries with answers outside the intersection, if the answer is *unknown*, the answer mechanism will answer correctly. For the remainder of answers lying outside the intersection, the answering mechanism will not terminate. For example, given the negative loops in the program segment *P2* above, the evaluation of either of the queries, $ans(p, X)$ and $ans(q, X)$, will execute without termination.

One might view a specification with multiple answer sets as incomplete, in the sense that there is not enough information in the specification to determine the value of a query that can have different values in different answer sets. The extended logic semantics of such an incomplete specification consists of all possible models that are consistent with the specification. The answering mechanism we present gives a definite answer to a query only if all answer sets contain the same answer for the query, and for cases where there is just one answer set, the answering mechanism corresponds to the theoretical semantics of the logic.

In software development, we would like our specifications to provide answers to all queries of interest to the users and developers. For this reason, we believe that it is always the goal of the specifier to construct a specification that is complete in the sense of having a unique answer set, at least for the set of queries that have practical value. However, it is almost certain that the initial versions of the specifications for any real system will not be complete in this sense. A theoretically complete answering mechanism for extended logic generates all possible answer sets even if there is more than one. Such an answering mechanism might be useful in practice for diagnosing incompleteness and helping developers understand what choices have to be made in order to make their specifications complete.

The answering procedure we present in program 3 is capable of detecting when the problems of inconsistency or incompleteness occur. Inconsistencies arise when the program's specification is no longer valid, i.e., the specification no longer correctly reflects the program context. Incompleteness means that the specification does not completely cover the program context. Incompleteness and inconsistency are the two fundamental problems of validation. Validation is the process by which we attempt to the answer the question: is the specification complete and consistent with respect to the environment. If a specification is not complete or consistent adaptive maintenance is necessary – the specification is, in other words, invalid.

All of the previous answers (for queries a - h) are unaffected by the change to the answer mechanism, but the answer to query (i) is now "inconsistency". In query (i), it can be observed that the basis of deciding who is and who is not a honors student has changed, as reflected in the inconsistency in the modified specification and program. Thus, the context shift implicit in the new statement about Sam is detectable via the new answering mechanism. The detection of such an inconsistency will lead a requirements engineer to propose one or more new specifications, check and modify the proposals until a validated new specification S_i is reached, and construct the corresponding program P_i to reflect the new context C_i.

The remainder of this paper shows how to combine the CAPS model and the answering mechanism.

3 Detecting Context Shifts in CAPS Prototypes via Logic Programming

CAPS is a computer aided prototyping system for real-time software (see [13] and [11]). CAPS is based upon a prototyping language PSDL (Prototype System Description Language, [12]). This section shows how a simulator for a small subset of PSDL can be realized in Prolog and augmented with the answering mechanism introduced in Section 2.4. We then use a simplified version of the missile defense example introduced by Lehman to illustrate how this approach can detect context shifts during the execution of the prototype.

3.1 A Spartan PSDL Simulator In Prolog

The fundamental constructs of PSDL are *operators, data streams, timing constraints, control constraints (i.e., condition and data triggers),* and *timers*. In this section, we present a simplified version of the CAPS system as implemented in Prolog. In particular, the simplified version implements the *operator, data streams,* and *control constraints*. To simplify the presentation, we have left out timing constraints, timers, user-defined data types, exceptions, and output guards. These features can be added without invalidating our approach. However, doing so would complicate the example considerably without contributing much to the issue of detecting context shifts.

A PSDL program is a network of operator and data streams, augmented by control constraints. We represent this network using assertions of the following form.

```
operator(Operator_Name, Input_Stream_List, Output_Stream_List).
```

The state of a PSDL computation can be described by giving the current data values on all of the data streams, and stating whether or not each of those values is new (i.e. whether or not it has been written since the last time it was read). We represent this information using assertions of the following form.

```
stream(Stream_Name, Current_data_value, New_or_not).
```

The designer specifies an application by writing a set of assertions of these forms, describing the network and the initial state of the data streams, as well as some assertions describing the intended behavior of the prototype, as described after we introduce the simulator.

The simulator is invoked via the `run` predicate, which specifies how many steps the simulator is to execute. In each step, the simulator scans all of the operators, and fires those that are ready to execute based on the triggering conditions specified in the user-defined network of operators. When an operator fires, it reads a value from each input stream and writes a value on each output stream. The core of the simulator is shown below.

```
/* Mini-PSDL Simulator. */
run(0).   /* The simulation is complete when Steps = 0. */
run(Steps) :- scan_operators, New_Steps is Steps - 1, run(New_Steps).

scan_operators :- get_op(Name), operator(Name, Inputs, Outputs),
  nl, print(executed(Name)), nl, fire(Name, Inputs, Outputs).

/*  Random choice on operators for nondeterminism   */
get_op(Name):-choose(operator, Name).


fire(Name, [I | Other_Inputs], Outputs) :- stream(I, V, _),
  retract(stream(I, V, _)), asserta(stream(I, V, not_new)),
  print(read_from(Name, V, I)), nl, !, fire(Name, Other_Inputs, Outputs).
  /* Simulate reading the input streams in this clause. */

fire(Name, [], [O | Other_Outputs]) :- choose(O, V),
  retract(stream(O, _, _)), asserta(stream(O, V, new)),
  print(wrote_into(Name, V, O)), nl, !, fire(Name, [], Other_Outputs).
  /* Simulate writing into the output streams in this clause. */

fire(_, [], []). /* Operator execution complete. */

/* Random Choice Functions. */

choose(What, Choice) :-
   last_choice(What, N), random(1, N, K), choice(What, K, Choice).
  /* Pick the K-th declared choice for random K, */
  /* 1 =< K =< number of choices available. */
choose(What, Choice) :- choose(What, Choice).
  /* Try a different choice if a failure backtracks to a choose. */
```

The simulator prints out a trace of all the operators that were executed, all of the input values that were read from data streams, and all of the output values written into the data streams. Note that changes to the state of the prototype are caused by both reading from input streams (the value in the stream is no longer new) and by writing to output streams (the stream gets a new value). The operation of the simulator is based on a mixture of random sampling (for output values) and semantic constraints on output values (provided by the descriptions of operator behavior in the `choice` declarations). The nondeterminism inherent in the CAPS model is reflected in the random selection of operators for execution. The `choose` predicate makes a random choice from a set of possibilities defined using assertions of the following form.

```
choice(What_Attribute_To_Choose, Index_Number, Value_To_Choose).
```

The `choose` predicate is used for choosing the operators (in `get_op`) and for choosing the values that an operator writes into its output data streams. The choices can be made deterministic by supplying computation rules in the declarations of the choices to add semantic information, as illustrated by the examples in Section

3.2. A more general explanation of the use of the simulator is given after we introduce the examples. The rest of the simulator consists of a pseudo-random number generator and the answering mechanism presented in Section 2.4. A complete listing of the simulator can be found in Appendix A. The simulator does not contain negation as failure (NFR). Thus it has a unique answer set and, therefore, implements the Extended Logic Semantics.

3.2 The Missile Firing Problem

We illustrate the definition of a prototype in the notation of the logic programming PSDL simulator using an example adapted from a problem introduced by [10], the missile defense system for an allied vessel. Immutable specifications in this domain are likely to include the following: (1) shoot down all hostile missiles, (2) descriptions of radar signatures for particular kinds of missiles, etc. A simplified version of this example can be represented as the network of operators defined by the following declarations and illustrated graphically in Fig. 4.

```
operator(radar, [], [detected_missile]).
operator(radio, [], [has_hit_ally]).
operator(intelligence_database,
        [detected_missile, has_hit_ally, hostile_missiles],
        [threat, hostile_missiles]) :- stream(detected_missile, _, new).
operator(defense_system, [threat], [fire_control]) :-
        stream(threat, Threat, new), Threat == true.
operator(defensive_weapon, [fire_control], []) :-
        stream(fire_control, shoot, new).
operator(completeness_monitor, [threat], []) :-
        stream(threat, unknown, _).
operator(consistency_monitor, [threat], []) :-
        stream(threat, inconsistency, _).

choice(operator, 7, consistency_monitor).
choice(operator, 6, completeness_monitor).
choice(operator, 5, defensive_weapon).
choice(operator, 4, defense_system).
choice(operator, 3, intelligence_database).
choice(operator, 2, radio).
choice(operator, 1, radar).
```

The operator declarations specify the network by listing the operators in the network and by listing the inputs and outputs of each operator. Triggering conditions are also defined by the right hand sides of the rules, as explained in Section 3.4.

The operators `completeness_monitor` and `consistency_monitor` are not part of the application – they are included to monitor the prototype execution for evidence of context shifts. The operator `completeness_monitor` is triggered whenever the `threat` stream carries the value `unknown` and the operator `consistency_monitor` whenever the `threat` stream carries the value `inconsistency`. These are the two special values produced by our extended answering mechanism when it cannot determine a normal Boolean answer. This answering

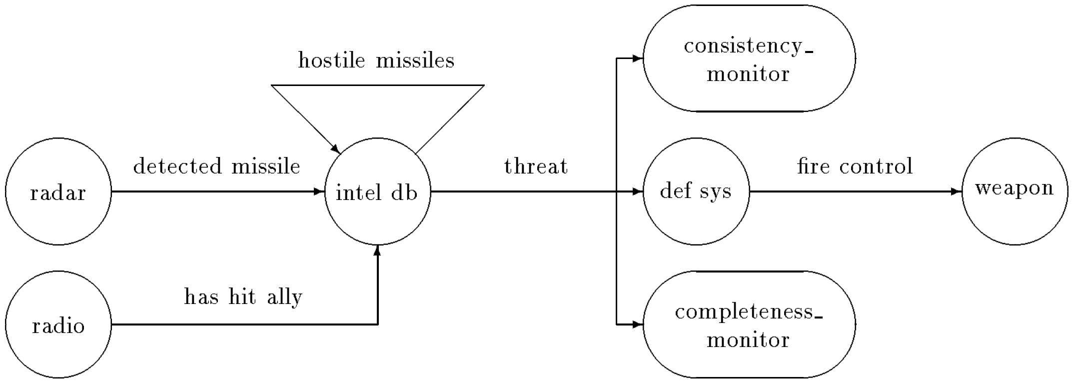


Figure 4: The Missile Defense Prototype

mechanism is provided as part of the PSDL simulator, and is explained in Section 2.4. The purpose of these operators is to alert the designer to situations in which context shifts have affected some of the assumptions on which the specification is based - if everything works as expected, then these operators should never be invoked. In general, such operators should be placed on every stream whose value is defined using the `ans` predicate (e.g., the stream `threat`). This rule can be used to automatically place such monitoring operations if we use a translator program to generate the Prolog declarations from a PSDL definition of a prototype.

We also have to specify the initial states of the data streams. For the example, the initial states are:

```
/* Stream declarations: stream(Name, Current_data_value, New_or_not). */
stream(detected_missile, patriot, not_new).
stream(has_hit_ally, scud, not_new).
stream(hostile_missiles, [], not_new).
stream(threat, none, not_new).
stream(fire_control, none, not_new).
```

The behavior model of a prototype is defined using `choice` declarations. The behavior definitions for the example follow.

```
/* Behavior model. */
choice(detected_missile, 5, phantom2).
choice(detected_missile, 4, phantom1).
choice(detected_missile, 3, patriot).
choice(detected_missile, 2, exocet).
choice(detected_missile, 1, scud).

choice(has_hit_ally, 5, phantom2).
choice(has_hit_ally, 4, phantom1).
choice(has_hit_ally, 3, patriot).
choice(has_hit_ally, 2, exocet).
choice(has_hit_ally, 1, scud).

choice(threat, 1, X) :- stream(detected_missile, M, _), ans(not(friendly(M)), X).
choice(hostile_missiles, 1, X) :- stream(hostile_missiles, L, _), stream(has_hit_ally, M, _),
                                  include(M, L, X).
choice(fire_control, 1, X) :- stream(threat, T, _), fire_control_policy(T, X).
```

```
include(X, L, L) :- member(X, L). /* include(X, Y, Z) => Z = {X} U Y */
include(X, L, [X | L]).
```

The PSDL simulator supports two kinds of behavior simulation. The first is a rough model that works by random sampling from a set of declared possible values for a stream. The example shows this kind of definition for the stream `detected_missile`, which is produced by the `radar` operator, and for the stream `has_hit_ally`, which is produced by the operator `radio`. In this case we are simulating data from external sensors by random sampling.

The second kind of behavior modeling is more detailed, and computes the resulting data values according to specified rules. For example, the stream `threat` produced by the intelligence_database operator is modeled deterministically and in detail; the value produced by the only possible choice is represented by the value of a variable, which is computed according to a stated rule: the stream `threat` is supposed to carry the Boolean value true if and only if the detected missile (from the the input stream of the `intelligence_database` operator) is classified as not friendly. The `intelligence_database` operator also maintains a state variable in the stream `hostile_missiles`, which is a list that contains all of the missiles that have been reported as having hit an allied vessel. The `fire_control` operator operates according to a simple fire control policy: shoot down a missile if it is hostile. The definitions for policies like these are part of the context model associated with the prototype. The context model for the example is shown below.

```
friendly(exocet).
friendly(patriot).
not(friendly(scud)).

not(friendly(M)) :- stream(has_hit_ally, M, _).
not(friendly(M)) :- stream(hostile_missiles, L, _), member(M, L).

member(X, [X | _]). /* Set membership test. */
member(X, [_ | L]) :- member(X, L).

fire_control_policy(Threat, shoot) :- Threat == true.
fire_control_policy(Threat, do_not_shoot) :- Threat \== true.
```

The context model indicates which missiles are known to be friendly, which are known to be hostile, and how a missile can be determined, dynamically, to be hostile. This determination is based on the contents of the intelligence database, which summarizes and keeps track of the intelligence reports that are modeled by the data stream `has_hit_ally`. Notice, based upon the framework introduced in section 1.3:

1. The specifier notices that the specifications concerning which missiles are hostile or friendly are actually mutable;

2. The specifier indicates rules which would contradict the mutable specifications (i.e., if a missile strikes an allied ship it is considered hostile);

3. The specifier identifies additional inputs (e.g., has_hit_ally) required in order to monitor the software context;

4. The answering mechanism provides the ability to discover inconsistencies when they arise.

In steps 2 and 3, intelligence reports on missile sales could be used to determine when missiles should be considered hostile. Such information would prevent any ship from being sunk.

Notice that the PSDL simulator with the extended answering mechanism could be applied to any number of possible prototypes. In other words, the example illustrates a general approach that can be applied to detect context shifts in many different kinds of applications.

3.3 An Example of Detecting Context Shifts

We now examine five sample prototype executions. In the first, a patriot is fired into the vessel's airspace. Given the information that the patriot is friendly (i.e., friendly(patriot) in the original specification) and given no additional information about the patriot, the ship does not shoot the patriot (i.e., the patriot is not viewed as a threat):

```
         :
executed(radar)
wrote_into(radar, patriot, detected_missile)

executed(radio)
wrote_into(radio, scud, has_hit_ally)

executed(intelligence_database)
read_from(intelligence_database, patriot, detected_missile)
read_from(intelligence_database, scud, has_hit_ally)
         :
wrote_into(intelligence_database, false, threat)
         :
```

In the next example, a scud (a missile known to be hostile) is fired and another scud hits an allied vessel. In this case the system shoots the scud:

```
         :
executed(radar)
wrote_into(radar, scud, detected_missile)
         :
executed(radio)
wrote_into(radio, scud, has_hit_ally)
         :
wrote_into(intelligence_database, true, threat)
wrote_into(intelligence_database, [scud], hostile_missiles)

executed(defense_system)
read_from(defense_system, true, threat)
wrote_into(defense_system, shoot, fire_control)
         :
```

Next, a phantom1 missile is fired into the vessel's airspace and a scud hits an allied vessel. There is no information concerning whether the `phantom1` is friendly or hostile. In other words the software context is presenting information which shows that the current specification is incomplete:

```
         :
executed(radar)
wrote_into(radar, phantom1, detected_missile)

executed(radio)
wrote_into(radio, scud, has_hit_ally)
         :
executed(intelligence_database)
read_from(intelligence_database, phantom1, detected_missile)
read_from(intelligence_database, scud, has_hit_ally)
read_from(intelligence_database, [], hostile_missiles)
wrote_into(intelligence_database, unknown, threat)
wrote_into(intelligence_database, [scud], hostile_missiles)
         :
executed(completeness_monitor)
read_from(completeness_monitor, unknown, threat)
         :
```

This may be an indication of a context shift, especially if there was previously an assumption that the friendliness (or otherwise) of all possible missile types was known to the missile defense system. The `completeness_monitor` operator is executed to alert the designer that something unexpected has happened.

As an example of the application of the simulator to detecting context shifts, consider the situation in the missile defense example where it is known that an `exocet` has hit an allied ship and an `exocet` is fired.

```
         :
executed(radio)
wrote_into(radio, exocet, has_hit_ally)
         :
executed(radar)
wrote_into(radar, exocet, detected_missile)
         :
executed(intelligence_database)
         :
read_from(intelligence_database, [exocet], hostile_missiles)
wrote_into(intelligence_database, inconsistency, threat)
         :
executed(consistency_monitor)
read_from(consistency_monitor, inconsistency, threat)
         :
```

Recall that the `exocet` is considered `friendly` in the original specification, but the system has "sensed" information implying that the `exocet` is `not friendly`. The context (as sensed by the system) has shifted resulting in an inconsistency (which is detected via the answering mechanism extension to Prolog).

Finally consider how this framework also allows for the prototyped system to learn about its context. Recall that there is no information about the `phantom1` missile. Suppose a `phantom1` sinks an allied ship. Given this information it is reasonable to assume that the phantom1 missile is hostile. The following sequence of events causes the defense system to treat the phantom1 missile appropriately:

```
         :
executed(radio)
wrote_into(radio, phantom1, has_hit_ally)

executed(radar)
wrote_into(radar, phantom1, detected_missile)

executed(intelligence_database)
read_from(intelligence_database, phantom1, detected_missile)
read_from(intelligence_database, phantom1, has_hit_ally)
read_from(intelligence_database, [], hostile_missiles)
wrote_into(intelligence_database, true, threat)
wrote_into(intelligence_database, [phantom1], hostile_missiles)
:
executed(defense_system)
read_from(defense_system, true, threat)
wrote_into(defense_system, shoot, fire_control)
         :
```

The combination of the PSDL language and the answering mechanism of Section 2.4 results in a framework to produce prototypes which are capable of reporting when they are inconsistent or incomplete. In other words, given the appropriate inputs, a prototype developed in this framework can report when data occurs which indicates that the prototype is not valid with respect to its context. It can report on the validity of a system in terms of both completeness and consistency. We believe this to be a significant step towards the automation of software maintenance activities. Also notice that in this example it is a minor step which leads to a fault tolerant response to the detected inconsistencies. For example, the system could be arranged in a manner where if an inconsistency arises, the missile causing the inconsistency is shot down. Fault tolerant responses could be provided for incomplete knowledge as well. The potential for fault tolerance that this work implies is the subject of continued research.

3.4 The General Approach

This section outlines how a prototyping project would use the PSDL simulation framework presented in Section 3.1. First, the author of a prototype would define the prototype as a network of operators by writing a set of operator declarations. In the network declarations for the PSDL simulator, PSDL operators without guard conditions are represented as Prolog assertions of the following form.

```
operator(Name, Input_streams, Output_streams).
```

In the example, the radar operator has this form. In general, operators with PSDL control constraints of the form TRIGGERED BY ALL x, y IF p(x, y) are represented as Prolog rules of the following form.

```
operator(Name, Input_streams, Output_streams) :-
  stream(x, X, new), stream(y, Y, new), p(X, Y).
```

This rule says that *both* streams `x` and `y` must have new values, and that these values must satisfy the execution guard `p(X, Y)`. In the example, the `defense_system` operator has this form. The `defensive_weapon` operator illustrates an equivalent simplified representation for this kind of guard, and the `intelligence_database` operator illustrates the form of the rule when there is a data trigger without any execution guard. This rule works because the `scan_operators` procedure will only find those operators whose execution guards are true in the current state.

PSDL control constraints of the form `TRIGGERED BY SOME x, y IF p(x, y)` are represented as sets of rules of the following form.

```
operator(Name, Input_streams, Output_streams) :-
  stream(x, X, new), stream(y, Y, _), p(X, Y).
operator(Name, Input_streams, Output_streams) :-
  stream(x, X, _), stream(y, Y, new), p(X, Y).
```

The above rules say that *at least one* of the streams `x` and `y` must have new values, and that values on these streams must satisfy the execution guard `p(X, Y)`. This form of the rules is illustrated by the `intelligence_database` operator in the example, which is triggered by either a new detected missile or a new report of a missile hitting an ally. These alternatives are randomly determined in the `choice` predicates.

The prototype designer must also supply definitions of the initial values for any data streams representing state variables, and define the behaviors of the operators, which is characterized by the data values each operator can write into its output streams. This is done either by explicitly listing a set of legal values, or by giving rules for computing those values. The designer must define a context model that is sufficient to support the rules for computing output values. The context model usually contains facts known to affect the decisions made by the operators and rules which allow the system to discover dynamically when facts may change.

Note that the developer must not only analyze the problem solution, but also must determine what information is needed to detect when the proposed problem solution is no longer valid. The analysis of the missile defense system recognizes that the decision to fire is based upon whether or not an incoming missile is friendly. It has also considered which missiles are known to be friendly in the original software context and how the system may discover if some missile is later found to be unfriendly. In the defense system example, the author needed to add the rule that a type of missile is to be considered hostile if a missile of that type has hit an allied vessel. Based on this rule, it was determined that the `has_hit_ally` information needed to be reported to the system to enable the PSDL framework to detect context shifts automatically.

Providing such information places a new burden on the analyst. In addition to determining the solution to a problem in terms of a system, the analyst must also determine what information the system will require in order for it to determine when the system should be modified. In other words, the analyst must specify the software solution and also design its maintenance process from the start.

4 Conclusions and Summary

The extended prototyping framework suggests the following process model for software evolution:

1. Divide a system into its immutable and mutable specifications.
2. Determine what information must be reported to the system so that it can analyze itself to determine if it needs to be modified.
3. Validate the original specifications via rapid prototyping. This step is iterated based upon effective interaction with the client.
4. Develop the system release based upon final prototype.
5. Verify the system release according to previously validated specifications.
6. Maintain the software in a proactive rather than reactive manner.

In step 6, we propose that maintainers of a software system periodically review all specifications to determine which are immutable and which are mutable (because of the possibility of misclassifications) and revalidate the mutable specifications, since these are the specifications which are most likely to become invalid over the life of the project. The mutable specifications are periodically revalidated with emphasis on those recently reclassified as mutable. Any mutable specification found to be no longer valid triggers an exploration of changes to the specifications.

Prior to changing the production software, the proposed change is validated together with the entire system of specifications to ascertain side effects of the change. This is accomplished via prototyping. Thus the effect of change becomes more predictable and manageable. Once the change has been validated through the prototype, the requirements tracing feature can be used to identify the program unit(s) which need(s) to be adapted to incorporate the validated change into the released software.

CAPS suggests a form of maintenance which is more scientific, proactive, and focused than many of the approaches currently used in practice. Although CAPS is targeted at real-time systems, CAPS and the general philosophy of the CAPS framework should facilitate the management of uncertainty in any context.

5 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*, 13:1-18, 1990.

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

[3] C. Baral and M. Gelfond, "Logic Programming and Knowledge Representation", *Journal of Logic Programming*, Vol. 19, No. 20, 1994, pp.73-148.

[4] C. Baral and V. Subrahmanian, *Duality between alternative semantics of logic programs and nonmonotonic formalisms*, in the *International Workshop in logic programming and nonmonotonic reasoning*, ed. Nerode, Marek and Subrahmanian (MIT press, 1991), pages 69-86.

[5] V. Berzins, Luqi, A. Yehudai, "Using Transformations in Specification-Based Prototyping", *IEEE Transactions on Software Engineering*, May 1993, pp. 436-452.

[6] B. Boehm, "A Spiral Model of Software Development and Enhancement", *Computer*, Vol. 21, No. 5, May, 1988, pp. 61-72.

[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*, pp. 1070–1080, (Seattle, Washington, August 15-19, 1988).

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

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

[10] M. Lehman, "Keynote Address", *IEEE CASE '90 Fourth International Workshop on Computer Aided Software Engineering*, December 5-8, Irvine California.

[11] Luqi, and M. Ketabchi, "A Computer-Aided Prototyping System", *IEEE Transactions on Software Engineering*, October 1988.

[12] Luqi, V. Berzins, and R. Yeh, "A Prototyping Language for Real-Time Software", *IEEE Transactions on Software Engineering*, October 1988.

[13] Luqi and D. Cooke, "The Management of Uncertainty in Software Development", *IEEE COMPSAC '92*, Chicago, IL, pp. 381–386.

[14] J. McCarthy, *Circumscription - A Form of Non-Monotonic Reasoning*, in *Artificial Intelligence*, 13:27–39 (1980).

[15] D. McDermott, Nonmonotonic Logic II: Non-monotonic Modal Theories. *J. ACM*, 29:33–57, 1982.

[16] H. Mills, "Software Development", *IEEE Trans. on Software Eng.* SE-2, 4 (Dec, 1976), 265–273.

[17] R. Moore, *Semantical considerations on nonmonotonic logic*, in *Artificial Intelligence*, 25:75–94 (1985).

[18] 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.

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

[20] R. Reiter, "An Sound and Sometimes Complete Query Evaluation Algorithm for Relational Databases with Null Values", *CACM*, 33(2): 349–370.

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

A Complete Listing of the Spartan PSDL Simulator

```
/* Mini-PSDL Simulator. */
:- dynamic stream/3, seed/1. /* Declarations for assert, retract. */

run(0).   /* The simulation is complete when Steps = 0. */
run(Steps) :- scan_operators, New_Steps is Steps - 1, run(New_Steps).

scan_operators :- get_op(Name), operator(Name, Inputs, Outputs),
  nl, print(executed(Name)), nl, fire(Name, Inputs, Outputs).

/*  Random choice on operators for nondeterminism   */
get_op(Name):-choose(operator, Name).


fire(Name, [I | Other_Inputs], Outputs) :- stream(I, V, _),
  retract(stream(I, V, _)), asserta(stream(I, V, not_new)),
  print(read_from(Name, V, I)), nl, !, fire(Name, Other_Inputs, Outputs).
  /* Simulate reading the input streams in this clause. */

fire(Name, [], [O | Other_Outputs]) :- choose(O, V),
  retract(stream(O, _, _)), asserta(stream(O, V, new)),
  print(wrote_into(Name, V, O)), nl, !, fire(Name, [], Other_Outputs).
  /* Simulate writing into the output streams in this clause. */

fire(_, [], []). /* Operator execution complete. */
```

```
/* Random Choice Functions. */

choose(What, Choice) :-
   last_choice(What, N), random(1, N, K), choice(What, K, Choice).
  /* Pick the K-th declared choice for random K, */
  /* 1 =< K =< number of choices available. */
choose(What, Choice) :- choose(What, Choice).
  /* Try a different choice if a failure backtracks to a choose. */

last_choice(What, N) :- choice(What, N, _).
  /* Find the last index for a choice. */
  /* Choices are numbered consecutively from 1 to N. */

random(L, U, N) :- L =< U, newseed(S), N is (S mod (1 + U - L)) + L.
  /* N is a pseudo-random number in the range L =< N =< U. */

newseed(S) :- seed(Old_Seed), S is (125 * Old_Seed + 1) mod 4096,
  retract(seed(Old_Seed)), asserta(seed(S)).

seed(45). /* Initial value of the seed. */

/* Logic Programming Extensions. */

not(not(P)) :- P.

ans(P, inconsistency) :- P, not(P).
ans(P, true) :- P.
ans(P, false) :- not(P).
ans(_, unknown).

not_known(P) :- P, !, fail.
not_known(_).
```

B Complete Listing of the Missile Defense Example

```
operator(radar, [], [detected_missile]).
operator(radio, [], [has_hit_ally]).
operator(intelligence_database,
       [detected_missile, has_hit_ally, hostile_missiles],
   [threat, hostile_missiles]) :- stream(detected_missile, _, new).
operator(defense_system, [threat], [fire_control]) :-
   stream(threat, Threat, new), Threat == true.
operator(defensive_weapon, [fire_control], []) :-
   stream(fire_control, shoot, new).
operator(completeness_monitor, [threat], []) :-
   stream(threat, unknown, _).
operator(consistency_monitor, [threat], []) :-
   stream(threat, inconsistency, _).

/* Stream declarations: stream(Name, Current_data_value, New_or_not). */
stream(detected_missile, patriot, not_new).
stream(has_hit_ally, scud, not_new).
stream(hostile_missiles, [], not_new).
```

```
stream(threat, none, not_new).
stream(fire_control, none, not_new).

/* Behavior model. */



choice(detected_missile, 5, phantom2).
choice(detected_missile, 4, phantom1).
choice(detected_missile, 3, patriot).
choice(detected_missile, 2, exocet).
choice(detected_missile, 1, scud).

choice(has_hit_ally, 5, phantom2).
choice(has_hit_ally, 4, phantom1).
choice(has_hit_ally, 3, patriot).
choice(has_hit_ally, 2, exocet).
choice(has_hit_ally, 1, scud).

choice(operator, 7, consistency_monitor).
choice(operator, 6, completeness_monitor).
choice(operator, 5, defensive_weapon).
choice(operator, 4, defense_system).
choice(operator, 3, intelligence_database).
choice(operator, 2, radio).
choice(operator, 1, radar).


choice(threat, 1, X) :- stream(detected_missile, M, _), ans(not(friendly(M)), X).
choice(hostile_missiles, 1, X) :- stream(hostile_missiles, L, _),
  stream(has_hit_ally, M, _), include(M, L, X).

choice(fire_control, 1, X) :- stream(threat, T, _), fire_control_policy(T, X).

/* Context model. */
fire_control_policy(Threat, shoot) :- Threat == true.
fire_control_policy(Threat, do_not_shoot) :- Threat \== true.

friendly(exocet).
friendly(patriot).
not(friendly(scud)).

not(friendly(M)) :- stream(has_hit_ally, M, _).
not(friendly(M)) :- stream(hostile_missiles, L, _), member(M, L).

member(X, [X | _]). /* Set membership test. */
member(X, [_ | L]) :- member(X, L).

include(X, L, L) :- member(X, L). /* include(X, Y, Z) => Z = {X} U Y */
include(X, L, [X | L]).
```