Current logic programming systems, as typified by PROLOG, contain limitations which restrict their usefulness during the specification, design and testing of software. A major limitation is the inability to perform analysis in the presence of incomplete information. Three sources of incompleteness are discussed here. First, the analysis is incomplete because the system is only partially finished. Second, in order to provide overall guidance, the analysis is first performed at an abstract level. The abstraction can be done selectively in order to focus the analysis. Third, some forms of incompleteness can only be resolved at run time by examining the properties of objects which are determined dynamically. It is the program itself which resolves the last form of incompleteness.

In logic programming, the program is expressed in terms of predicates relating the input and output and execution proceeds by constructing a proof of these input/output relationships. Generic Constraint Logic Programming (GCLP) is a form of logic programming developed to address incompleteness in analysis. 1 Predicates, which cannot be proven because of incompleteness, are treated as constraints which must be satisfied in order for the relationship to hold. The set of objects which satisfy these constraints form a subset of the domain for which the relation is true. The satisfaction of constraints is considered domain specific and is not handled directly by the theorem prover. Thus GCLP system is only one part of an analysis system which also includes a knowledge base, a strategist which focuses control and rules for constraint satisfaction. In the presence of incompleteness, the theorem prover returns a list of unsatisfied constraints. The strategist may resubmit the constraints to the theorem prover using a new set of axioms selected from the knowledge base.

The following specification fragment illustrates some of the above points.

\[
\text{not(within}(C,P1),\text{within}(C,P2),\text{within}(C,P3)).
\]

where C is a circle (with a known radius R) and P1, P2 and P3 are points. This rule states that there exist three points which cannot be contained in a circle of known radius R. The center of the circle, if it exists, is unknown but its position is constrained by the three within predicates. For the condition to be true, there must be no circle which can satisfy all three constraints simultaneously. In order to complete this specification, knowledge from several domains (geometry, logic, algebra) must be used. Assume that background knowledge in these domains already exists. However, even at this level of abstraction, the system can be analyzed. The rule:

\[
\text{not}(A,B,C) :- \text{not}(A,B)
\]

where A, B and C are bound to the three within predicates, focuses the analysis on the special case of finding two of the three points which cannot be contained in the circle. (Notice, the use of second order logic to reason about predicates). This special case is much easier to solve than the general case. Suppose that in the domain knowledge about geometry, the following rule is available:

\[
\text{not(within(Circle,Point1),within(Circle,Point2)) :- distance(Point1,Point2,Dist), diameter(Circle,Diam),greater(Dist,Diam)).
\]

This rule states that it is impossible to contain two points within a circle if the distance between the two points is greater than the diameter of the circle.

The result of this analysis is interesting for several reasons. First, it suggests an efficient algorithm for the special case. Furthermore, it partitions the problem for testing purposes into three classes, 1) those points which satisfy the special case, 2) those which satisfy the general case but not the special case and 3) those points which don't satisfy either. Thus, the analysis has uncovered possible structure within the implementation including the case in which the implementation has failed to address class 2 correctly (error of omission) but handles classes 1 and 3 correctly.

This analysis was performed without deciding how these geometric objects might be represented (a partial design). Even after deciding representation, proving that the greater relation holds between the distance and the diameter must be deferred until run time. If the constraint is refined to the point of containing only primitives in the underlying programming language, then it is an implementation.