Testing Production System Programs

Grigoris Antoniou
School of Computing & Information Technology
Griffith University
Brisbane, QLD 4111, Australia
ga@cit.gu.edu.au

Oliver Jack
Fachbereich Elektrotechnik
Universität-GH Paderborn
D-33095 Paderborn, Germany
oliver@adt.uni-paderborn.de

Abstract

A production system (PS) is a forward chaining rule-based system used to build large expert systems. Testing a PS must involve the construction of a covering set of test data but it is not clear what the meaning of covering a PS is and how a test data set can be measured according to coverage. We propose a test data coverage measure for a subset for PS with well defined semantics. We use a correspondence between PS and function free first order Horn logic programs to define the declarative coverage notion and measure. We found that the coverage measure can be used to determine the coverage of the program logic of a PS as well as to automate test data generation. Unification theory is utilised to measure test data coverage and constrained inductive generation is used for test data construction.

1 Introduction

A *production system* is a forward chaining rule-based system [5]. Such systems are used to build large expert systems for diverse domains, including troubleshooting in telecommunication networks, computer configuration systems. These expert system reason with large quantities of data. Hence such systems are implemented using database technology, the use *extensional databases* [4, 12].

Production systems have been implemented with integrating first order logic programs and function-free first order relational databases [3]. Testing a production system program (PS program) involves techniques developed for testing logic programs [7]. We consider a implementation-oriented testing approach. The aim is to thoroughly test the program which requires a test adequacy criterion [13]. A PS program has a declarative semantics defined via a corresponding logic program as defined by Raschid et al. [10], and testing can be performed on this logic program instead of the PS program. The corresponding logic program describes the logic of the PS program. A implementation-oriented test should then cover the program logic. This is considered the test adequacy criterion. To apply the criterion, the coverage of a test input set for a program must be defined and measure has to be provided to determine test coverage. The implementation-oriented test involves a two-step approach depicted in Figure 3.

1. The first step is the transformation of the PS program into its corresponding logic program to obtain the program logic. This can be done mechanically using the method described by Raschid et al. [10]
2. The second step is to apply test inputs to the logic program and measure the test coverage. This is described in the following sections.

We propose a declarative test coverage measure that can be utilised for automated test input generation. Test coverage is measured using a dual of the operational execution model for logic programs [8]. This provides guidelines for test input generation.

We refer to a test adequacy criterion based on the structure of the program logic. Ruggieri [11] devel-

oped a testing method for logic programsbased on the formal semantics of a logic program. The approach is to identify a subset of logic programs with decidable semantics, such that it can be tested whether a program entails a given finite set of atoms. This is a specification-oriented testing approach, but it is restricted to a certain class of logic programs. The corresponding logic program to a PS program can be outside this class of programs.

The present paper is part of ongoing work on the verification and validation of knowledge-based systems, which has attracted recent attention (for example, see [1]); it is organised as follows. Section 2 describes PS programs and the conditions for providing declarative semantics defined by corresponding logic programs. Section 3 presents our implementation-oriented testing measure, and section 4 contains our method for automated coverage-driven test input generation. Section 5 summarises the results and contains concluding remarks. Figure 3 and Tables 1 and 2 are found at the end of the paper.

2 Production System Programs

A *production rule* consists of a antecedent on the left hand side (the *body*) of the rule, the symbol $\rightarrow$ and a consequent on the right hand side (the *head*). The body is a conjunction of first order positive literals of the form $p(\bar{u})$ or negative literals of the form $\neg q(\bar{v})$. The heads (also called *actions*) are of the form $\texttt{assert}(r(\bar{u}))$ or $\texttt{retract}(s(\bar{v}))$. p, q, r, and s are predicates corresponding to database relations of the same arity, and $\bar{u}$ and $\bar{v}$ are vectors of terms from a nonempty set of constants a set a variables. All variables are *range-restricted*, i.e., for each production rule, each variable occurring in a literal must also occur in a positive literal in the body.

Here are function-free production rules considered. A function-free *production rule program* consists of

- a set of *a-productions*, which have a single `assert` action in the head
- a set of *r-productions* which have a single `retract` action in the head
- an initial extensional database of ground atoms

The operational semantics of a production rule program is the following:

The body of each production rule is interpreted as a query against the database relations. First, the positive literals are queried and the variables occurring in them are instantiated by matching atoms of the database. Then the negative literals, which are now ground due to range restriction of the variables, are queried against the database. The body of a production rule is *satisfied* if the relation contains instantiated tuples corresponding to each of the positive literals and if the relations do not contain tuples corresponding to the negative literals.

A production system program may not terminate and the relations may be updated infinitely. For example consider an initial database with two facts, $P(a)$, $Q(a)$, and the production rules

$P(X), Q(X) \rightarrow \texttt{assert}(R(X))$
$R(X) \rightarrow \texttt{assert}(S(X))$
$P(X), S(X) \rightarrow \texttt{assert}(T(X))$
$R(X), T(X) \rightarrow \texttt{retract}(R(X))$

The first three production rules will execute and the facts $R(a)$, $S(a)$, $T(a)$ will be added to the database. Then, by the fourth production rule, $R(a)$ is deleted and the first and fourth production rules will execute infinitely. To prevent this cycling effect, a production system program must possess a certain property, called stratified.

A function-free production system program PS with productions rules of the form

$$(A_1, \ldots, A_k, \neg B_1, \ldots, \neg B_l \rightarrow \texttt{assert}(P)),$$

$$(A_1, \ldots, A_k, \neg B_1, \ldots, \neg B_l \rightarrow \texttt{assert}(P))$$

is *stratified* [10], if there exists a partition $PS = PS_0 \cup PS_1 \cup \cdots \cup PS_n$, where PS_i are pairwise disjoint, such that for $i = 0, \ldots, n$ the following holds:

1. For every predicate A_k occurring positively in the body of a production rule in PS_i, all a-productions where A_k occurs in the `assert` action, must be included in $\bigcup_{j \leq i} PS_i$.

2. For every predicate A_k occurring positively in the body of a production rule in PS_i, all r-productions where A_k occurs in the `retract` action must be included in $\bigcup_{j<i} PS_i$.

3. For every predicate B_k occurring negatively in the body of a production rule in PS_i, all a-productions where B_k occurs in the `assert` action must be included in $\bigcup_{j<i} PS_i$.

4. For every predicate B_k occurring negatively in the body of a production rule in PS_i, all r-productions where B_k occurs in the `retract`-action must be included in $\bigcup_{j<i} PS_i$.

5. For the predicate P of the `retract`-action of an r-production rule in PS_i, all production rules with P occurring in the head must be included in $\bigcup_{j\leq i} PS_i$.

The partition PS_0 comprises the initial database and contains no production rules.

A function-free stratified PS program possesses a corresponding logic program [10], hence it is semantically equivalent to its corresponding logic program. The corresponding logic program defines the formal semantics of a PS program. The testing approach presented in this paper relates to the formal semantics of PS programs. In the subsequent sections we assume that the program to be tested is stratified.

An example expert system domain of a classification of different fire types and means to extinguish them is given in Table 1. An excerpt of the PS program for the fire type classification and extinguishing is given in Table 2. The relevant part of the program comprises the classification of fire types. The guidelines for extinguishing are presented for some examples. The corresponding logic program looks as follows: almost every production rule is essentially a logic program rule. For example, the first production rule is translated into the logic program rule

$$\text{OrdinaryCombustible}(X) \leftarrow \text{Paper}(X).$$

The only exception has to do with the production rule involving `retract`. Its meaning is that in case it can be concluded that X is both type A and type B, we retract type A to treat the fire as type B. This is achieved by adding a new precondition to the logic program rule corresponding to the fourth production rule in the system given in Table 2 (it is the only rule which can assert that a fire is of type A). The logic program rule looks as follows:

$$\text{typeA}(X) \leftarrow \text{Burning}(X), \text{OrdinaryCombustible}(X), \neg\text{typeB}(X).$$

A formal description of constructing the corresponding logic program is given by Raschid [10].

3 Declarative Testing Measure

Systematic testing of a PS programs must involve the measurement of the test. Measurement means test coverage determination, which is a quantification of the test method. Quantification is necessary to obtain test data adequacy [13]. Test data adequacy criteria are used to derive or define stopping rules for testing. There are two categories of test data adequacy criteria

1. specification-based testing, which specifies the required testing from the specification

2. implementation-based testing, which specifies the required testing from the program

With the first criterion, a test data set is adequate, if all the identified features of the specification have been fully exercised. In the second criterion, a test data set is adequate, if the program in its internal structure has been thoroughly exercised.

Thoroughly exercising a program requires the quantification of testing. Such a quantification is a test coverage measure that quantifies the amount to which a program is exercised by a set of test data.

We use a PS program as the only source for defining the test, hence implementation-based testing takes place. A PS program is a predicate logic-based description. The declarative semantics of a PS program is defined by a corresponding logic program [10]. We will build our test coverage measure upon this logic program. With procedural programming, the program logic is implicit, while with logic programming the program comprises the program logic itself. Here,

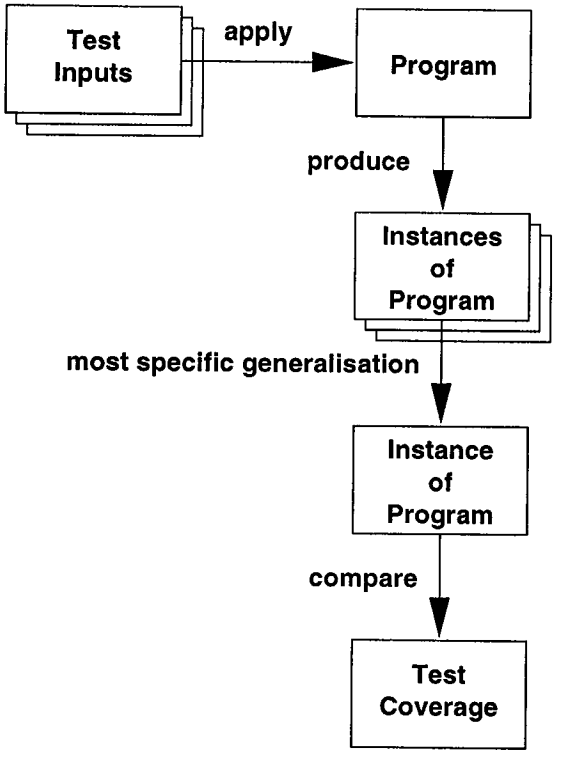

Figure 1: Test coverage

covering the program means covering the program logic. Conventional test coverage measures such as statement coverage, branch coverage and path coverage are not applicable to both PS programs and logic programs, since they rely on control flow which is not explicitly expressed.

The test coverage measure is defined on the basis of the following observation [7, 2]: A logic program specifies a problem solution in form of a set of general facts and rules. A test input (query) to a logic program generates a specialisation of the facts and rules. It generates an *instance* of the program by providing an answer to a specific query, using the general facts and rules. Generalising the instances generated by the test inputs reconstructs partly the facts and rules of the program. The generalisation is itself an instance of the program. A program is then *covered* by a set of test inputs if the generalisation of the instances reconstructs the whole program. The concept of test coverage is depicted in Figure 1. The above given informal description of our test coverage approach is formally presented in the following.

A logic program consists of a finite set of rules (also called *program clauses*) of the form

$$H \leftarrow L_1, \ldots, L_m,$$

where H is an atom called the *head* and $L_1, \ldots, L_m$ is called the *body*. $i = 1, \ldots, m$. In case $m = 0$ the rule a *fact*. The head is a positive literal of the form $p(\bar{t})$ and the body is a conjunction of positive literals (also called *atoms*) of the form $q(\bar{s})$ or negative literals of the form $\neg q(\bar{s})$. The symbols p and q are predicates, and the arguments $\bar{t}$, respectively $\bar{s}$, are terms $t_1, \ldots, t_n$, respectively $s_1, \ldots, s_n$. Since terms, atoms and clauses share the same syntactical the are addressed as expressions. An *expression* is a term, an atom or a clause.

The expression E is an *instance* of the expression F, denoted by $E \leq F$, if E is obtained by F through substitution of variables $v_1, \ldots, v_k$ occurring in F by terms $t_1, \ldots, t_k$. This is denoted by $E = F\sigma$, where $\sigma = \{v_1/t_1, \ldots, v_k/t_k\}$. Two expressions E and F are *equivalent*, denoted $E \equiv F$, if $E \leq F$ and $F \leq E$. Equivalent expressions differ only in the naming of the variables occurring in them. The expression E is a *strict instance* of the expression F, denoted by $E < F$, if $E \leq F$ but not $E \equiv F$. The set of expressions is augmented by two elements, $\top$ and $\bot$. The element $\top$ is the greatest element ($E \leq \top$ for every expression E) and $\bot$ is the least element ($\bot \leq E$ for every expression E). The set of expressions with the instance relation forms a complete lattice, which means that for every subset S of expressions there exists a greatest lower bound $\sqcup S$ and a least upper bound $\sqcap S$. Computing the greatest lower bound of a set of expressions is called *unification* and computing the least upper bound is called *anti-unification*. There exist efficient algorithms for computing greatest lower bounds [9] and least upper bounds [6] of finite sets of expressions. If for a set S of expressions $\bot < \sqcap S$, then S is called *unifiable*. For a unifiable set $S = \{E_1, \ldots, E_k\}$ of expressions there exists a sub-

stitution μ such that $\sqcap S = E_i\mu$ for $i = 1, \ldots, k$. The substitution μ is called the *most general unifier* of S.

Testing a logic program involves as test input a *goal* which is an atom to be unified with the head of program clauses. A goal is an input to a logic program P. It represents a query Q which is answered by the set of instances of Q that P entails. A goal is denoted by $(\leftarrow Q)$, where Q is an atom. The goal $G = (\leftarrow Q)$ generates a set $C \downarrow G$ of instances of program clauses in the following way. If Q and the head H of the program clause $C = (H \leftarrow L_1, \ldots, L_m)$ are unifiable then $C\mu = (H\mu \leftarrow L_1\mu, \ldots, L_k\mu)$, where μ is the most general unifier of $\{H, Q\}$, is contained in the set $C \downarrow G$. The notion of goal-generated clause instances extends to sets of goals. For a set $T = \{G_1, \ldots, G_n\}$ of goals and a program clause C,

$$C \downarrow T := \bigcup_{i=1,\ldots,n} C \downarrow G_i$$

is called the *instantiation set* of C by T. It represents specialisations of C given by the test input goals T and contains information about the way which the test inputs access the clause C.

The most specific generalisation of the instantiation set is said to be the *coverage* of C by T. The most specific generalisation is the least clause $\tilde{C}$ (with respect to the instantiation relation) such that each element of $C \downarrow T$ is an instance of $\tilde{C}$; thus, $\tilde{C} = \sqcup(C \downarrow T)$. The coverage $\sqcup(C \downarrow T)$ is always an instance of the program clause C since $C \downarrow T$ contains only instances of C. If $C \downarrow T \equiv C$, then the program clause C can be reconstructed form the instances given by the test inputs T. In this case the test input set T is said to *cover* C, and T is called a *cover* for C.

The program clause coverage provides the basis for the definition of program coverage. A set T of test input goals generates instantiation sets of all program clauses $C_1, \ldots, C_m$ of the tested program P, and the most specific generalisations of these instantiation sets are instances of the program clauses. Hence, the test input goals T generate an instance

$$P \downarrow T := \{C_1 \downarrow T, \ldots, C_m \downarrow T\}$$

of P. The coverage of a test input set T for a program $P = \{C_1, \ldots, C_m\}$ is

$$\sqcup(P \downarrow T) := \{\sqcup(C_1 \downarrow T), \ldots, \sqcup(C_m \downarrow T)\}.$$

Note that if $C_i \downarrow T = \emptyset$ for an $i \in \{1, \ldots, m\}$, then $\sqcup(C_i \downarrow T) = \bot$, which means that none of the test inputs from T matched a clause of the program. The coverage identifies

1. untested program clauses ($\sqcup(C_i \downarrow T) = \bot$)
2. tested but not covered program clauses ($\bot < \sqcup(C_i \downarrow T) < C_i$)
3. covered program clauses ($\sqcup(C_i \downarrow T) \equiv C_i$)

4 Test Input Generation

The coverage notion can be used to automate test input generation. This is achieved by analysing the coverage $\tilde{C}$ of a program clause C. For a tested but not covered program clause C, its coverage $\tilde{C}$ is called the *covered instance* of C. The covered instance can be used to construct new test inputs in order to increase the coverage. The covered instance is an attempt to reconstruct the tested program clause from the test outputs. The reconstruction through most specific generalisation provides a "part" of the program clause, i.e. an instance of the program clause. The differences between the program clause and its covered instance can then be used to guide the generation of the next test input. For example let

$$C = (P(X, Y, Z) \leftarrow Q(X), R(Y, Z))$$

with a coverage

$$\tilde{C} = (P(W, W, a) \leftarrow Q(W), R(W, a)).$$

This coverage is obtained by the test inputs

$$T = \{(\leftarrow P(a, a, a)), (\leftarrow P(b, b, a))\}.$$

The coverage $\tilde{C}$ means that C is tested for identical first two arguments and the third argument a. Selecting a test input with the first two arguments being different will increase the coverage. For a test

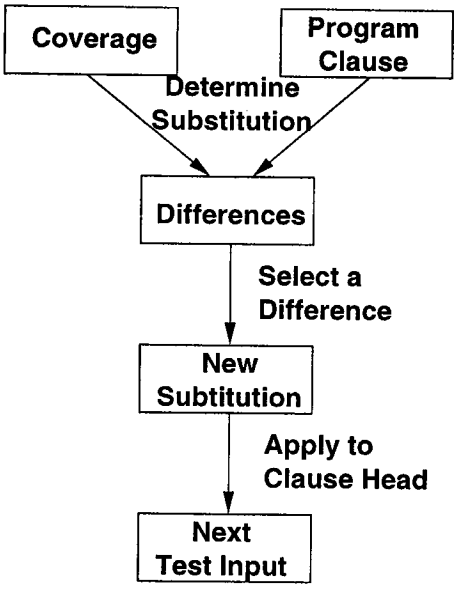

Figure 2: Construction of next test input

input $t = (\leftarrow P(a,b,a))$ the coverage of $T \cup \{t\}$ for C is $\hat{C} = (P(W,V,a) \leftarrow Q(W), R(V,a))$, and, since $\tilde{C} < \hat{C}$, the coverage increases. The construction of a test input to increase the actual coverage is depicted in Figure 2.

The coverage $\tilde{C}$ of C is an instance of C, and there exists a substitution $\sigma = \{V_1/t_1, \ldots, V_k/t_k\}$ such that $\tilde{C} = C\sigma$, where V_i are variables occurring in C for $i = 1, \ldots, k$. If C is not covered, then σ is not a variable renaming. Then at least one of the following properties holds for σ:

Difference type 1 There exists a pair V_i/t_i such that t_i is not a variable

Difference type 2 For V_i/t_i and V_j/t_j with $i,j \in \{1,\ldots,k\}$ and $i \neq j$, the terms t_i and t_j are identical variables.

The substitution for the coverage $\tilde{C}$ in the example above is $\sigma = \{X/W, Y/W, Z/a\}$. We can select one of the differences from σ and construct a new substitution. For example, we select the difference $(X/W, Y/W)$. Then two different constants, a and b, are chosen to substitute these for X and Y. This new substitution $\tilde{\sigma} = \{X/a, Y/b, Z/a\}$ is applied to the head of C to obtain the new test input $t_1 = (\leftarrow P(a,b,a))$. The new coverage of $T \cup \{t_1\}$ for C is $\tilde{C}_1 = (P(W,V,a) \leftarrow Q(W), R(V,a))$.

Now, there is only one difference, Z/a. We choose a constant b, different from a, and select some arbitrary terms a and b to be substituted for X and Y. Then the resulting substitution $\sigma_1 = \{X/a, Y/b, Z/b\}$ is applied to the head of C to obtain the next test input $t_2 = (\leftarrow P(a,b,b))$. The coverage of $T \cup \{t_1, t_2\}$ for C is now $\tilde{C}_2 = (P(W,V,U) \leftarrow Q(W), R(V,U))$, and clause C is covered.

The generation of test inputs is *coverage-driven* and *constrained* by the differences analysed from the substitution σ which describes the "distance" of the actual coverage to the tested clause.

5 Summary

Testing of rule-based declarative programs, such as production systems programs, involves test adequacy criteria based on declarative programming concepts. Since production systems are strongly related to first order logic programs, testing concepts for logic programming are applicable to testing of production systems. A production system program, as well as a logic program, expresses directly the program logic. We propose an implementation-oriented testing approach and define a test adequacy criterion through a test coverage measure that relates to covering the program logic. The test coverage measure is the dual to the execution model of logic programming. It defines the coverage as the generalisation of the test outputs. This generalisation might end up in a rule that is equivalent to the tested program rule, which means that the program rule is covered.

Our test coverage adequacy criterion enables automated test input generation. The test inputs are generated incrementally form the actual test coverage. This is achieved by reconstructing partly the tested program rules from the actual test outputs. The differences between the reconstructed rules and the corresponding program rules guide the test input generation.

The testing method relates to the formal semantics of production system programs. Not every production system program possesses a well-defined formal semantics. The class of production system program with well-defined formal semantics is restricted to stratified programs. Hence our testing method is only applicable to this class of programs. The programs outside the class of stratified programs include non-terminating programs and programs with ambiguous outputs to the same inputs. It is a future task to extend the presented testing approach to general production system programs.

References

[1] G. Antoniou and R. Plant, editors. *Verification and Validation of Knowledge-Based Systems.* Technical Report WS-97-01. AAAI Press, Menlo Park, 1997.

[2] F. Belli and O. Jack. A test coverage notion for logic programming. In *Proceedings of the Sixth International Symposium on Software Reliability Engineering*, pages 133–142, Toulouse, France, October 1995.

[3] J. Bocca. EDUCE: A marriage of convenience: Prolog and a relational DBMS. In *Symposium on Logic Programming*, pages 36–45. IEEE Computer Society, The Computer Society Press, Sept. 1986.

[4] L. M. L. Delcambre and J. N. Etheredge. A self-controlling interpreter for the relation production language. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 17(3):396–403, Sept. 1988.

[5] F. Hayes-Roth. Rule-based systems. *Communications of the ACM*, 28(9):921–932, Sept. 1985.

[6] G. Huet. *Resolution d'equations dans des languages d'ordre* $1, 2, \ldots, \omega$. These d'etat, Universite de Paris VII, 1976.

[7] O. Jack. *Software Testing for Conventional and Logic Programming*, volume 10 of *Programming Complex Systems.* Walter de Gruyter & Co., Berlin, New York, 1996.

[8] J.-L. Lassez, M. Maher, and K. Marriott. Unification revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kauffman, 1988.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin Heidelberg, 1987.

[10] L. Raschid. A semantics for a class of stratified production system programs. *The Journal of Logic Programming*, 21(1):31–57, Aug. 1994.

[11] S. Ruggieri. Decidability of logic program semantics and applications to testing. In H. Kuchen and S. D. Swierstra, editors, *Programming Languages: Implementations, Logics, and Programs, Proceedings of the 8th International Symposium, PLILP '96*, Lecture Notes in Computer Science, pages 347–362, Aachen, Germany, Sept. 1996. Springer-Verlag.

[12] E. Simon, J. Kiernan, and C. de Maindreville. Implementing high level active rules on top of a relational DBMS. In L.-Y. Yuan, editor, *18th International Conference on Very Large Data Bases*, pages 315–326, Vancouver, Canada, 23–27 Aug. 1992. Morgan Kaufmann.

[13] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

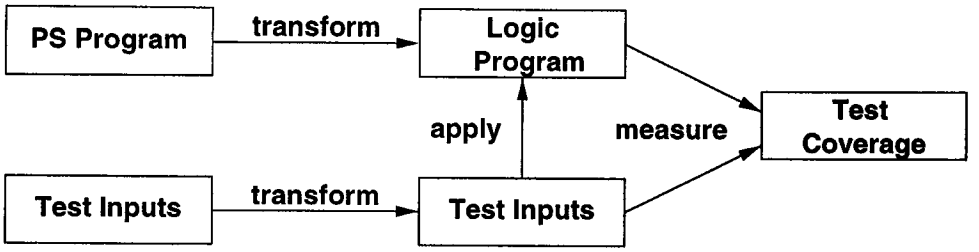

Figure 3: Implementation-oriented test of a PS program

Fire Classification		
Type	**Material**	**Extinguish**
A	involve ordinary combustibles such as paper, wood and cloth	with heat-absorbing or combustion-retarding extinguishers such as water or water-based liquids and dry chemicals
B	flammable and combustible liquids (such as oils and gas), greases, and similar material	by excluding air, inhibiting the release of combustible vapors, or interrupting the combustion chain reaction. Extinguishers include dry chemicals, carbon dioxide, foam, and bromotrifluoromethane
C	energized electrical equipment	should be extinguished with a non-conducting agent to prevent short circuits. If possible the poser should be cut. Extinguishers include dry chemicals, carbon dioxide, and bromotrifluoromethane
D	combustible metals such as magnesium and sodium	with smothering and heat-absorbing chemicals that do not react with the burning metals. Such chemicals include trimethoxyboroxine and screened graphitized coke
If the fire could be either Type A or Type B, treat as Type B		

Table 1: Example expert system domain of fire classification and extinguishing

Paper(X) → **assert**(OrdinaryCombustible(X)))
Wood(X) → **assert**(OrdinaryCombustible(X))
Cloth(X) → **assert**(OrdinaryCombustible(X))
Burning(X), OrdinaryCombustible(X) → **assert**(typeA(X))
Oil(X) → **assert**(Flammable(X))
Oil(X) → **assert**(CombustibleLiquid(X))
Gas(X) → **assert**(Flammable(X))
Gas(X) → **assert**(CombustibleLiquid(X))
Burning(X), flammable(X), CombustibleLiquid(X) → **assert**(typeB(X))
Burning(X), Grease(X) → **assert**(typeB(X))
Burning(X), ElectricalEquipment(X), Energized(X) → **assert**(typeC(X))
Magnesium(X) → **assert**(CombustibleMetal(X))
Sodium(X) → **assert**(CombustibleMetal(X))
Burning(X), CombustibleMetal(X) → **assert**(typeD(X))
typeB(X) → **retract**(typeA(X))
typeA(X) → **assert**(UseHeatAbsorbing(X))
...
typeB(X) → **assert**(ExcludeAir(X))
...
typeC(X) → **assert**(NonconductingAgent(X))
...
typeD(X) → **assert**(Smothering(X))
...

Table 2: An example PS program