Detecting Redundant Production Rules †

James G. Schmolze
Dept. of Electrical Eng. and Computer Science
Tufts University
Medford, MA 02155 USA
schmolze@eecs.tufts.edu

Wayne Snyder
Dept. of Computer Science
Boston University
111 Cummington Street
Boston, MA 02215 USA
snyder@cs.bu.edu

Abstract

We present a general method for detecting redundant production rules based upon a term rewrite semantics. We present the semantic account, define rule execution over both ground memories and memory schemas, and define redundancy for production rules. From those definitions, an algorithm is developed that detects redundant rules, and which improves upon previously published methods.

Production rule (PR) systems (Hayes-Roth 1985) have found their way into many applications over the past decade (see, e.g., (Mockler 1992)). One limitation to their expanded use, especially in safety critical operations, is the difficulty of validating and verifying (V&V) their behavior (Hayes-Roth 1985). One aspect of V&V for production rules is the detection of redundant rules, which is important for at least four reasons. (1) During rule development, when rules are changing, making a change to a rule will not have the proper impact if its redundant counterpart remains in force. (2) Rule sets often execute more efficiently without redundancies. (3) For systems that combine evidence by examining multiple paths from a start state to a conclusion, redundant rules lead to extraneous paths. (4) A test for redundant rules can form the basis of high-level testing of rule sets since the test for whether or not a rule set R can compute Q from P can be rephrased as a test for whether or not the rule $P \Rightarrow Q$ is redundant with respect to R.

Most previous papers on detecting redundant PRs use pattern matching methods. Namely, they provide patterns of two or three rules that, if matched onto a portion of a given rule set, imply that one or more of the matched rules are redundant.

We take a different approach. We present an operational semantics for PR systems (PRSs) that views PR execution as a rewriting process, specifically term rewriting (TR). We then use this account to define rule redundancy formally and to develop a general method for detecting wide classes of redundant rules. In our approach, detecting redundant rules becomes a search problem, yielding a method that subsumes and improves upon existing published methods. Moreover, we are able to analyze some previously published methods to check their correctness with respect to our semantic account.

Our Model of Production Rule Systems

The following is our model of PRSs. An *atom* is an expression of the form $P(t_1, \ldots, t_n)$, where P is some predicate symbol and the t_i are terms; a *literal* is an atom or the negation of an atom. Unlike some production systems, we allow function symbols in terms. An atom or literal is *ground* if it contains no variables. We assume that the signature is infinite – i.e., that there are an infinite number of constants (e.g., the real numbers). A PRS consists of a *working memory* (WM), which is a set of ground atoms, and a *production memory* (PM), which is a set of rules. When P is ground and W is a WM, we say that P is true with respect to W iff $P \in W$ and that $\neg P$ is true with respect to W iff $P \notin W$. Thus, negation is defined as a simple form of the *closed world assumption*.[1] A rule has the form

$$r : C_1, \ldots, C_n \Rightarrow A_1, \ldots, A_m$$

with the following features:

1. Each side of the rule, called *left-hand side* (LHS) and *right-hand side* (RHS) respectively, is a set.
2. Each condition C_i is either an atom, testing for the presence of atoms in the WM, or a negated literal, testing for the absence of atoms.
3. Each action A_i is of the form "ADD A" or "REMOVE A" for some atom A.

   Often, such rules of this form are called *pure*.
4. For each A_i of the form "REMOVE A", there is some unnegated $C_j = A$.

†Thanks to the reviewers for their detailed comments.

[1] Ginsberg & Williamson (1993) call this *negation as absence*. This contrasts to *negation as failure* in which $\neg P$ is true iff one cannot *prove* P.

5. All variables appearing in the rule must appear in at least one unnegated C_i.[2]
6. There are no two actions of the form "ADD A" and "REMOVE A'" where A and A' can unify.

Restriction 6 means that we do not handle rules such as $P(x), Q(y) \Rightarrow$ REMOVE $P(x)$, ADD $P(y)$, which can cause inconsistencies. In such a case, when $x = y$, this rule implies that $P(x)$ must be both added and removed.

A *substitution* is a mapping from variables to terms, e.g., $\{x \mapsto P, y \mapsto Q, \ldots\}$ and the application of a substitution σ to a term t is denoted $P\sigma$ or, equivalently, $(P)\sigma$. For example, if $P = f(x, h(y))$ and $\sigma = \{x \mapsto a, y \mapsto b, z \mapsto c\}$, then $P\sigma = f(a, h(c))$.

A rule, such as r above, *matches* against a WM W if and only if (iff) there exists a ground substitution σ such that (1) for each unnegated C_i, $(C_i)\sigma \in W$, and (2) for each negated C_i, $(C_i)\sigma \notin W$. Each match of a rule is called an *instantiation*, which consists of the rule plus the substitution that led to the match. The result of executing an instantiation is the WM that results from executing the instantiated RHS. Due to restriction 6 above, the order of executing RHS actions does not matter.

When running, a PRS executes the following basic loop until the MATCH step yields zero instantiations.

1. MATCH: Determine all instantiations.
2. SELECT: Select an instantiation to execute according to the *conflict resolution strategy* (CRS). However, we do not model any particular CRS and assume, instead, that the CRS is non-deterministic but fair; i.e., no instantiation will remain unselected indefinitely.
3. ACT: Executes the selected instantiation.

Notation: We write $P \longrightarrow_R Q$ when R is a rule set, P and Q are WMs, and executing an instantiation of some rule in R from P produces Q. In such cases, we say that P *rewrites* to Q, or equivalently, that P *reduces*, or *simplifies*, to Q. We will leave out R when it is obvious from context. Let $\longrightarrow^*$ be the reflexive transitive closure of $\longrightarrow$, i.e., $P \longrightarrow^* Q$ iff $\exists P_0, \ldots, P_n$ such that $P = P_0 \longrightarrow P_1 \longrightarrow \ldots \longrightarrow P_n = Q$ for some $n \geq 0$. We say that a memory P is *final*, or equivalently *normal*, with respect to (wrt) P iff there is no Q such that $P \longrightarrow Q$. We write $P \longrightarrow^! Q$ as a shorthand for when $P \longrightarrow^* Q$ and Q is normal; in such cases, we say that Q is a *normal form* of P; a memory can have many normal forms. *Normalizing* a term means reducing it to a (possibly non-unique) normal form. Finally, a set of rules R is *terminating* iff there is no infinite sequence $P_1 \longrightarrow P_2 \longrightarrow P_3 \longrightarrow \ldots$.

We use four kinds of arrows in this paper. $\Rightarrow$ is used for production rules, $\rightarrow$ is used for term rewrite rules, $\longrightarrow$ is used for the rewrite relation computed by a given rule set, and $\mapsto$ is used to show the mappings in a substitution.

Background

First, we briefly review other possible methods for specifying a semantics for PRSs. Then we examine published methods for detecting redundant PRs. Beforehand, however, we note that since our rewrite semantics interprets negation very simply as the absence of a fact in the database, and not via an operational rule (e.g., negation as failure), that there is no strong connection to the various approaches to non-monotonic reasoning and to negation in logic programming.

Other Semantic Accounts

We do not model PR execution as deduction in a standard logic because, in the straight-forward translation, rule execution is non-monotonic and can lead to inconsistencies. For an example of non-monotonicity, take the rule "$A, -B \Rightarrow$ ADD C." Then $\{A\} \longrightarrow^! \{A, C\}$, which means that from A we can deduce C. But from $A \wedge B$, we cannot deduce C because $\{A, B\}$ is normal. For an example of inconsistency, take the rule "$A \Rightarrow$ REMOVE A." Then $\{A\} \longrightarrow^! \{\}$. In other words, from A, we can deduce $\neg A$.[3]

Raschid and Lobo offer both fixed-point and declarative semantic accounts for *strictly stratified* PR programs (Raschid 1994) and for *non-deterministic causal* (NDC) PR programs (Raschid & Lobo 1994). Strictly stratified PR programs always terminate and have unique normal forms. NDC programs always terminate but may have several normal forms. Both classes are strictly smaller than all PR programs. Groiss (1995) provides a declarative semantics that is defined in terms of Datalog (Ceri, Gottlog, & Tanca 1990). He uses a situation calculus (McCarthy & Hayes 1969) approach to define the semantics of *mOPS5*, which differs from OPS5 mainly by having a completely deterministic CRS, which he models fully. Both of these accounts can be used to determine the correctness of particular PR interpreters and for validating conditions on rule executions.

Waldinger and Stickel (1990) offer a declarative semantics for PRSs. They axiomatize set theory in first order logic and then encode production rule matching and execution in terms of set operations. Gamble *et al* (1994) offer an alternative account in terms of the SWARM system (Roman & Cunningham 1990). Both

[2] In (Snyder & Schmolze 1996), we show how to relax this restriction and allow for quantified negative testing. E.g., a rule like

$$Block(x), -On(y, x) \Rightarrow \text{ADD } Clear(x)$$

matches if there is a block x and no possible binding for y such that y is on x. We represent this with *constrained* rewrite rules, where the constraint captures, in this case, the implicit quantification on y. In this case, the constraint is $\forall y.\neg On(y, x)$.

[3] More restrictive models of PRs (e.g., (Ginsberg & Williamson 1993)) can be modeled by standard deduction.

accounts have been used for verifying conditions on rule sets and for deriving rule sets from specification.

However, none of the above accounts have the operational flavor that we need, and none have been used to define rule execution over memory schemas, which we have found extremely useful when detecting redundant rules and when performing certain V&V applications. Thus, we have turned to a rewrite semantics.

Related Work on Redundancy Detection

Our work is related to that of Sagiv (Sagiv 1988) on equivalence of Datalog programs; if we restrict our rules to the form "$A_1, \ldots, A_n \rightarrow$ ADD B", where all the A_i are positive, and use no function symbols, then our approach is equivalent to Sagiv's algorithm for *uniform containment*. Hence, our algorithm is effectively a generalization of his decision procedure for redundancy in Datalog programs.

Ginsberg and Williamson (1993) show a general method for detecting redundant rules for a restricted form of a PR system. An essential difference between their work and ours is that they do not allow REMOVE actions and can model rule execution as first-order deduction using an ATMS-based approach (de Kleer 1986).

Several researchers have examined rule redundancy for PR models similar to ours (e.g., (Suwa, Scorr, & Shortliffe 1982; Nguyen *et al.* 1987; Zhang & Nguyen 1994)) and have taken a pattern-based approach. Namely, they have defined a set of syntactic patterns that correspond to classes of redundant rules, and have provided algorithms that search through rule sets for these patterns. For example, Appendix A of (Lee & O'Keefe 1993) gives the first three patterns shown below (their format has been modified slightly). The idea is to see if a subset of the rules matches any of these templates and, if so, to eliminate the indicated redundant rule.

Class 1	*Class 3*
$P, Q \Rightarrow$ ADD S	$P, Q \Rightarrow$ ADD S
$Q, P \Rightarrow$ ADD S	$P \Rightarrow$ ADD S
Class 2	*Class 4*
$P, Q \Rightarrow$ ADD S	$P \Rightarrow$ ADD S
$S \Rightarrow$ ADD T	$Q \Rightarrow$ ADD T
$P, Q \Rightarrow$ ADD T	$P, Q \Rightarrow$ ADD S, ADD T

- Class 1: Simple redundancy. Either rule can be eliminated.
- Class 2: Redundancy in chaining. The last rule can be eliminated.
- Class 3: Subsumption. The second rule subsumes the first, and so the first rule can be eliminated.

We will not examine the algorithms used to test whether a subset of a rule set matches a given template; the details are fairly straight-forward.

However, we note the obvious, namely that this approach can only detect a limited set of redundancies, i.e., those represented by a template. Class 4, where the last rule is redundant, is an example that escapes detection by Classes 1–3. Of course, one could simply add this template to the collection, but the process is seemingly endless, adding templates in an *ad hoc* manner. Instead, we will offer a sound but incomplete procedure (that always terminates for terminating rule sets) that will test for redundancy in a systematic way.

A Brief Introduction to Term Rewriting Systems

A substitution σ is said to be a *most general unifier* of the terms S and T, i.e., $\sigma = mgu(S, T)$, if for any substitution θ such that $S\theta = T\theta$ (i.e., θ unifies S and T), there exists an η such that $\theta = \sigma\eta$ (where $x\sigma\eta$ is defined as $(x\sigma)\eta$). A substitution ρ is said to *match* S onto T if $S\rho = T$.

A *rewrite rule* is an expression $S \rightarrow T$, where S and T are terms. We write $P[P']$ to (ambiguously) denote that P' is a subterm of P, and say that P *rewrites to* the term Q using the rule $S \rightarrow T$ if S can be matched onto a subterm P' of P, i.e., $P = P[P'] = P[S\sigma]$, and $Q = P[T\sigma]$. In other words, the result of rewriting is the term Q that results from removing the matched term T and replacing it by $T\sigma$. For example, the rule $h(x, y) \rightarrow f(x)$ can rewrite $g(a, h(z, b))$ into $g(a, f(z))$. This is the normal "substitution of equals for equals" of algebra. It is usually assumed that in a rule $S \rightarrow T$, every variable in T occurs also in S. Rewriting is also called *reducing* or *simplifying*. We will use the notation $S \longrightarrow T$, $S \longrightarrow^* T$ and $S \longrightarrow^! T$ as defined earlier. See (Huet 1980) for a summary of the basic theory of term rewriting.

Mapping Production Systems into Term Rewrite Systems

Production systems can be thought of as term rewriting systems if we change our focus from rewriting terms to rewriting sets of literals. Our translation scheme comes from (Snyder & Schmolze 1996).

Translating Working Memories into Terms

The *working memory* (WM) of a production system is a finite set of ground atoms $\{A_1, \ldots, A_n\}$. This set forming operation can be considered to be a term $A_1 + A_2 + \ldots + A_n$ where $+$ is an associative, commutative, and idempotent (i.e., such that $A + A = A$) function symbol. Thus we could formally consider production systems as rewriting systems for such terms; however, we shall prefer set notation, and shall consider production systems as rules that rewrite sets.

In order to use the term rewrite framework, all literals on the left-hand side (LHS) of a rule must explicitly match a term in memory. This is true even for negated literals such as in the following rule.

$$On(x, y) \Rightarrow \text{ADD } Above(x, y)$$

To accomplish this, we make explicit all the negated literals in a memory. We therefore will consider each working memory to be "completed" with negative literals, following the *closed world assumption* (CWA).

Definition 1 *An* extended working memory *(EWM) is a set of ground literals W such that the number of positive literals in W is finite, and is said to be* consistent *iff for each ground atom A in the language, exactly one of A or $-A$ is in W.*

Thus each working memory W has a unique maximal consistent extension $W^+ = W \cup \{-A \mid A \notin W$ and A ground$\}$, which is a consistent EWM.

In what follows, we shall consider each unnegated literal to have a sign of $+$ and each negated literal to have a sign of $-$. However, we will often write unnegated literals without any sign; i.e., A is the same as $+A$.

Definition 2 *A* memory schema *is a set of literals that may use variables where the number of positive literals is finite, where signs are either $+$, $-$ or a variable, and where variables denoting signs do not appear elsewhere in the set of literals. The denotation of $M = \{L_1, ..., L_n\}$ is defined as follows: $\Phi(M) = \{W | W$ is a consistent EWM and $\exists \sigma . M\sigma \subseteq W\}$. A memory schema M is* consistent *iff $\Phi(M)$ is not empty.*

Thus, each EWM is a memory schema that has no variables. From now on, when we refer to *memories*, we mean memory schemas. Also, literals with a variable sign have no impact on the meaning of a term. For example, the denotation of $\{P, v\,Q\}$, where v is a variable sign not appearing in P nor Q, is exactly that of $\{P\}$. This leads to the notion of an *expansion.*

Definition 3 *Let M be a memory schema, A be an atom and v be a variable that does not appear in either M nor A. $M \cup \{v\,A\}$ is a* one-step expansion *of M via $v\,A$ iff both $M \cup \{+A\}$ and $M \cup \{-A\}$ are consistent. A memory schema M' is an* expansion *of M iff there is a series of one-step expansions from M to M'. A* one-step contraction *is the opposite of a one-step expansion, and a* contraction *is the opposite of an expansion.*

If M' is an expansion of M, then $\Phi(M') = \Phi(M)$.

With negated literals represented explicitly and with the use of memory schemas, we can represent memories in the same language as the LHSs of rules. For example, the rule above matches at least once in any EWM satisfying the schema $\{On(w, z)\}$. This capability will prove to be extremely useful.

Translating Production Rules into Term Rewrite Rules

Definition 4 *A* preserving rewrite rule *has the form*

$$v_1\,A_1, \ldots, v_n\,A_n \longrightarrow w_1\,A_1, \ldots, w_n\,A_n$$

where:

- *Each A_i is an atom, possibly containing variables.*
- *Each v_i is a $+$, a $-$ or a variable not appearing elsewhere in the rule.*
- *Each w_i is a $+$ or a $-$.*
- *Every non-sign variable also occurs in some positive literal on the LHS.*
- *For at least one i, $v_i \neq w_i$.*

In other words, the set of atoms is the same on each side, but possibly the sign of a particular atom has changed. We call such a rule *preserving* because it preserves the CWA property – i.e., each atom appears exactly once in a working memory, either negated or unnegated. The literals whose signs change will be called *active* and the others *passive*.

For example, the following PR

$$R_{on} : On(x, y) \Rightarrow \text{ADD } Above(x, y)$$

can be represented by the preserving rule

$$r_{on} : On(x, y), v\,Above(x, y) \rightarrow On(x, y), Above(x, y).$$

All production rules in our model can be formalized by a preserving rewrite rule.

Executing Production Rules

We now develop the notion of rewriting as needed to formalize the action of PRSs on WMs. This type of rewriting is called *simplification* (Huet 1980). We assume in what follows that any EWM mentioned is consistent (and in general infinite) and all rules are preserving. Moreover, we let U and V each denote a set of literals, P, Q, S and T each denote single literals, W denote an EWM, M denote a memory schema, and R denote a set of rules.

Definition 5 *We say that a consistent EWM W* rewrites to, *or* simplifies to, *W' via a rule $U \rightarrow V$ iff there exists a substitution σ such that $U\sigma \subseteq W$, $W' = (W - U\sigma) \cup V\sigma$, and $W \neq W'$. If W rewrites to W' via a rule in R we write $W \longrightarrow W'$, as usual.*

Thus, when executing a rule, we remove the instantiated literals matched by the LHS and replace them by the instantiated RHS. However, the memory must be changed by the sign of at least one literal. It is easy to show that if W is a consistent EWM, R a set of preserving rules, and $W \longrightarrow^* W'$, then W' is a consistent EWM. This is because the only changes resulting from rewriting an EWM is simply to "flip" positives to negatives and/or vice-versa, and moreover, no inconsistencies can be introduced (thanks to condition 6 in the section describing our PR model). This corresponds to applying production rules that add/remove ground atoms to/from the WM.

For an example of rewriting (ground) EWMs with preserving rules, we use the rule r_{on} from above. Let $W_0 = \{On(a, b)\}$. r_{on} matches against W_0^+ (where W_0^+ is the completed form of W_0) with $\sigma = \{x \mapsto a, y \mapsto b, v \mapsto -\}$ and rewrites to W_1^+ where $W_1 = \{On(a, b), Above(a, b)\}$.

Definition 6 *We say that a memory schema M rewrites to M' via rule $U \rightarrow V$ iff the variables in M and in U do not overlap, there exists a substitution σ such that $U\sigma \subseteq M$, $M' = (M - U\sigma) \cup V\sigma$, and $U\sigma \neq V\sigma$.*

If a memory and a rule have overlapping variables, we rename the variables in the rule as needed. The last requirement, $U\sigma \neq V\sigma$, eliminates rewrites that never change the memory. We note that when rewriting memory schemas, we sometimes must expand a memory in order to obtain a match.

Now we show an example similar to the above, but applied to memory schemas. Let $M_0 = \{+On(a,u)\}$, where a is a constant and u is a variable. Here $\Phi(M_0)$ contains all consistent EWMs W where there exists a σ that maps u to a constant and where $(M_0)\sigma \subseteq W$. Hence, W_0^+ (from above) is a member of $\Phi(M_0)$. r_{on} does not match against M_0 because there is no term matching $v\,Above(x,y)$. However, we can expand M_0 to $M_0' = \{+On(a,u), w\,Above(a,u)\}$ where w is a *fresh* variable – i.e., a variable that does not appear elsewhere. Rule r_{on} matches against M_0' with $\sigma = \{x \mapsto a, y \mapsto u, v \mapsto w\}$ and rewrites to $M_1 = \{+On(a,u), +Above(a,u)\}$.

When matching the LHS of a rule r to a memory M, we contract M so there are no literals with variable signs, and treat all other variables in M like constants. Next, we match all the positive literals in r's LHS, which binds all the non-sign variables in r. Then, we match the negative literals and, finally, the literals with variable signs. If a literal with a variable sign from r, say $v\,P$, does not match in M, we try to expand M. Letting σ be the bindings produced so far by the match, we try to expand M to $M' = M \cup \{w(P)\sigma\}$ where w is a fresh variable. Note that $(P)\sigma$ will contain no variables from r. If M' is a legal expansion, we use it in place of M, match $v\,P$ against $w(P)\sigma$ (which is guaranteed to match with $v \mapsto w$), and continue with the remaining literals. Thus, *the maximum number of expansions we must consider in order to obtain all possible matches is n, where n is the number of literals from r's LHS that have a variable sign.*

We note that simplification uses *set matching*, i.e., more than one LHS condition of a rule can match a single literal in a memory. For example, given a memory $M = \{A, ...\}$ and a rule $r : B, C, ... \rightarrow ...$, there might be a σ representing a rule match where $A = B\sigma = C\sigma$.

The algorithm for performing simplification is a straight-forward adaptation of *AC term rewriting*, which is term rewriting modulo the associativity and commutativity of certain term-forming operators (Dershowitz & Jouannaud 1990).

A Semantics for Production Systems

Definition 7 *Given a production rule set R, whose translation to preserving rules is R', we say that the semantics of R is $\Phi(R')$ where $\Phi(R') = \longrightarrow^*_{R'}$ and where $\longrightarrow^*_{R'} = \{< M, M' > | M$ and M' are (ground) EWMs and $M \longrightarrow^*_{R'} M'\}$.*

For simplicity, we will ignore the difference between R and R' and will just use R where the context will determine whether we mean the original set of production rules or its translated set of preserving rules.

Given this semantics, we can now define equivalence between rule sets.

Definition 8 *Two rule sets R_1 and R_2 are equivalent, written $R_1 \equiv R_2$, iff $\Phi(R_1)$ equals $\Phi(R_2)$.*

We note that the meaning of a rule set is defined in terms of the ground EWMs that can be rewritten and not in terms of memory schemas.

We define the semantics of a PR set in terms of *reachability*, namely, the set of memories that can be reached from a given memory after zero or more rewrites. As a result, intermediate memories matter; we are not only interested in final memories.

There are two obvious alternatives to $\longrightarrow^*$ for defining the semantics of R that we might have used, namely, $\longrightarrow$ or $\longrightarrow^!$. We reject $\longrightarrow$ as being too specific and we reject $\longrightarrow^!$ because we are interested in environments where rule sets are being developed, and in particular, where rules might later be added. With the use of $\longrightarrow^!$, if a rule r is determined to be redundant and then more rules are added to R, r may no longer be redundant wrt this expanded rule set.

We note that we do not consider the *conflict resolution strategy* (CRS) when defining our semantics. In a sense, $\longrightarrow^*_R$ denotes the set of all terms reachable under all possible CRSs. To take the CRS into account makes both our model of PRs and, more importantly, the associated reasoning considerably more complex. For example, to take *recency* into account – i.e., execute the rule instantiation that matches the atoms most recently added to the WM – one must add to the model time stamps for every atom, and one must take time stamps into account when determining $\longrightarrow$. While this is not difficult for the ground case, it is not possible to determine which rule will execute next on a memory schema. After all, the schema denotes a possibly infinite set of memories, including all possible orderings of the time stamps. One would need to express domain constraints about the time stamps, and reason with those constraints, thus making reasoning considerably more complex.

Detecting Redundant Rules

Thanks to our semantics, we are able to define rule redundancy in a general way that allows a rule to be redundant wrt a set of other rules, not just wrt one or two other rules. From this definition, the development of a sound detection procedure is straightforward. In fact, the procedure has the happy property that the pattern based approaches described earlier are subsumed in a natural way simply by limiting the depth of search.

We define rule redundancy as follows.

Definition 9 *Let R be a set of rules where $r \in R$. r is redundant wrt R iff $(R - \{r\}) \equiv R$.*

Clearly this is the broadest possible definition of redundancy, since we can only remove a rule whose departure leaves the semantics of R unchanged.[4]

The following forms the basis for our algorithm.

Lemma 1 *Let r be the rule $U \rightarrow V$ and R be a set of rules where $r \in R$ and $R' = (R - \{r\})$. Then r is redundant wrt R if $U \longrightarrow^*_{R'} V$.*

Proof: We assume that $U \longrightarrow^*_{R'} V$ and will show that r is redundant wrt R. Surely $\longrightarrow^*_{R'} \subseteq \longrightarrow^*_{R}$ given the definitions of R and R'. So, to show $R' \equiv R$, we only need to show that $\longrightarrow^*_{R} \subseteq \longrightarrow^*_{R'}$. To do this, we take an arbitrary chain of rewrites made using R, namely, $M_0 \longrightarrow_R M_1 \longrightarrow_R M_2 \longrightarrow_R \ldots \longrightarrow_R M_k$, and we show that $M_0 \longrightarrow^*_{R'} M_k$. Using R', we can make the same chain except for those i where $M_{i-1} \longrightarrow_R M_i$ using the rule $r : U \rightarrow V$, which does not exist in R'. However, any of these rewrites must be a specialization of $U \longrightarrow^*_{R'} V$, which we assumed true. Therefore, $M_0 \longrightarrow^*_{R'} M_k$. □

In general, this leads us to consider a general redundancy test that is based on *reachability*. The idea in testing the redundancy of a rule $U \rightarrow V$ in R is to search for a chain of rewrites that constitute $U \longrightarrow^* V$ without using the rule $U \rightarrow V$. A simple algorithm searches in breadth-first manner the tree of all terms reachable from U using the remaining rules in R and using a *closed list* (Nilsson 1980) to handle cycles.[5] When R is terminating, our algorithm always terminates thanks to the following.

Theorem 2 *When R is terminating, the set of terms reachable from a given term is finite.*

Proof: The tree of terms reachable from U has finite depth since R is terminating, and thus has no infinite chains of rewrites. Also, the branching factor at any node is finite since there are a finite number of rules in R and there are only a finite number of ways that a term can be rewritten by a given rule. This latter observation comes from the fact that every variable in a rule, except for sign variables, must appear in a positive literal on the rule's LHS. Since any memory schema has only a finite number of positive literals, there are but a finite number of ways of binding the rule's non-sign variables to values. Also, there are only two ways of binding the sign variables to constants (+ or −) and only a finite number of expansions to consider. □

This is a powerful method that is capable of finding wide classes of redundant rules. Although this method is incomplete, it appears only to miss cases where a subset of the rules partitions the context surrounding the rule being tested, and in each case rewrites to the same result except for the context, which is preserved. For example, given two rules "$A, B \rightarrow -A, B$" and "$A, -B \rightarrow -A, -B$", then the rule "$A \rightarrow -A$" is redundant because any ground memory containing A will also contain either B or $-B$, and in both cases A can rewrite to $-A$. We will be examining this type of case analysis in future research.

In addition, this algorithm "scales down" nicely to the pattern-based methods such as those shown earlier. For example, classes 1 (simple) and 3 (subsumption) can be found by searching down only to depth one in the reachability tree, and redundancy via simple chaining can be found by searching down to depth two. We note that class 2 from (Lee & O'Keefe 1993) is not correct for our semantic account (see next section), and so it is not an example of simple chaining.

The complexity of the algorithm when R is terminating is exponential in both time and space. In general, a practical search method could be resource limited – e.g., use an appropriate search method subject to resource limitations such as maximum depth. In the case of resource failure, one could then assume that the failure to prove that a rule is redundant means that the rule is not redundant.

We note that redundancy wrt our semantic account may differ from redundancy wrt a given CRS. However, as we discussed earlier, we do not take the CRS into account.

Applying the Detection Method

Using this characterization, it is easy to show the correctness or incorrectness of the redundant rule patterns 1–4 shown earlier. Class 1 becomes the following set of preserving rules.

$$P, Q, v\,S \rightarrow P, Q, S$$
$$Q, P, v\,S \rightarrow Q, P, S$$

It is obvious that $\{P, Q, v\,S\} \longrightarrow \{P, Q, S\}$ using the second rule and that $\{Q, P, v\,S\} \longrightarrow \{Q, P, S\}$ using the first rule, so each rule is redundant and Class 1 is correct.

Class 2 is not correct with respect to our semantics – it is not the case that the last rule is always redundant. Here is the template in preserving rule format.

$$\begin{array}{lll} P,\ Q,\ v\,S & \rightarrow & P,\ Q,\ S \\ \quad\quad S,\ v\,T & \rightarrow & \quad\quad S,\ T \\ P,\ Q, \quad v\,T & \rightarrow & P,\ Q, \quad T \end{array}$$

Using only the first two rules, it is not possible to rewrite $\{P, Q, vT\}$ to $\{P, Q, T\}$. While it is possible to rewrite $\{P, Q, vT\} \longrightarrow \{P, Q, S, vT\} \longrightarrow \{P, Q, S, T\}$ using the first two rules, that differs from the last rule and so we cannot show redundancy according to our theory. And indeed, there is a counterexample: $\{P, Q, -S, -T\} \longrightarrow^* \{P, Q, -S, T\}$. This is possible with all three rules but not with only the first two.

We note that this class of redundancy may be correct for other semantic accounts; e.g., when there are no

[4] Those familiar with rewriting will note that if the signature were finite, then we would have to broaden our notion of redundancy somewhat; however the assumption of infinite signatures is natural in the context of PR systems.

[5] The best search algorithm depends on the application.

REMOVE actions, and thus rewriting is monotonic. However, Class 2 is also incorrect when the semantics of rule set R is $\longrightarrow_R^!$ (this is the semantic account we rejected earlier). Consider these three rules within a four rule set that also includes the following.

$$P, \quad -S \rightarrow -P, -S$$

In this four rule set, we can get $\{P, Q, -S, -T\} \longrightarrow \{P, Q, -S, T\} \longrightarrow \{-P, Q, -S, T\}$, where this last term is normal. This could not occur without the third rule.

Classes 3 and 4 are both correct, though we do not show it here. However, the important idea is that classes 1, 2 and 4 plus others would be subsumed by our approach with a maximum search depth of 3. A greater search depth would be able to cover many more classes of redundancies.

We note that we have a software tool that detects redundant PRs using breadth-first search. We used this tool to check all examples in this paper, plus others. We will report on this tool in another paper.

Summary

We have presented a formal definition of production rule redundancy and an algorithm for detecting redundant production rules. The algorithm is sound but not complete. Its complexity is exponential in the general case but always terminates when the rule set is terminating. The method has the happy property that by applying it with limited resources – namely, by limiting the depth of search – wide classes of redundant rules can be detected. Specifically, these classes include those that would be discovered by other published methods. Thus, this method both subsumes and improves upon these methods. Our method is based upon a semantic account of production rules based on term rewriting. This account and the formality of our approach not only inspired our algorithm but allowed us to check the correctness of other methods.

Using this semantic account, we have elsewhere shown how to test production rule sets for confluence (Snyder & Schmolze 1996). In the future, we will examine other V&V concerns, namely, detecting other types of anomalies, such as conflicting rules, and formal verification.

References

Ceri, S.; Gottlog, G.; and Tanca, L. 1990. *Logic, Programming and Databases.* Springer-Verlag.

de Kleer, J. 1986. An assumption-based TMS. *Artificial Intelligence* 28:127–162.

Dershowitz, N., and Jouannaud, J.-P. 1990. Rewrite Systems. In Leeuwen, J. v., ed., *Handbook of Theoretical Science.* Elsevier Science Publishers B.V.

Gamble, R. F.; Roman, G.-C.; Ball, W.; and Cunningham, H. C. 1994. The application of formal specification and verification to rule-based programs. *International Journal of Expert systems* 7(3):303–338.

Ginsberg, A., and Williamson, K. 1993. Inconsistency and Redundancy Checking for Quasi-First-Order-Logic Knowledge Bases. *International Journal of Expert Systems* 6(3):321–340.

Groiss, H. 1995. A Formal Semantics for mOPS5. In *Proceedings of the Seventh IEEE International Conference on Tools with Artificial Intelligence.*

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

Huet, G. 1980. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM* 27(4):797–821.

Lee, S., and O'Keefe, R. M. 1993. Subsumption Anomalies in Hybrid Knowledge Bases. *International Journal of Expert Systems* 6(3):299–320.

McCarthy, J., and Hayes, P. J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Michie, D., eds., *Machine intelligence 4.* New York: American Elsevier.

Mockler, R. J. 1992. *Developing Knowledge-Based Systems Using an Expert System Shell.* New York: Macmillan.

Nguyen, T. A.; Perkins, W. A.; Laffey, T. J.; and Pecora, D. 1987. Knowledge Base Verification. *AI Magazine* 8(2):69–75.

Nilsson, N. J. 1980. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga Publishing Company.

Raschid, L., and Lobo, J. 1994. A Semantics for a Class of Non-Deterministic and Causal Production System Programs. *Journal of Automated Reasoning* 12(3).

Raschid, L. 1994. A Semantics for a Class of Stratified Production System Programs. *Journal of Logic Programming* 21(1):31–57.

Roman, G.-C., and Cunningham, H. C. 1990. Mixed Programming Metaphors in a Shared Dataspace Model of Concurrency. *IEEE Transactions on Software Engineering* 16(12):1361–1373.

Sagiv, Y. 1988. Optimizaing Datalog Programs. In Minker, J., ed., *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann.

Snyder, W., and Schmolze, J. G. 1996. Rewrite Semantics for Production Rule Systems: Theory and Applications. In *Proceedings of the 13th International Conference on Automated Deduction.*

Suwa, M.; Scorr, A. C.; and Shortliffe, E. H. 1982. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine* 3:16–21.

Waldinger, R. J., and Stickel, M. E. 1990. Proving Properties of Rule-Based Systems. Technical Report Technical Note 494, SRI International.

Zhang, D., and Nguyen, D. 1994. PREPARE: A Tool for Knowledge Base Verification. *IEEE Transactions on Knowledge and Data Engineering* 6(6):983–989.