*International Journal of Innovative Computing, Information and Control*
Volume **10**, Number **5**, October **2014**

ICIC International ©2014 ISSN 1349-4198
pp. 1653–1667

# DETERMINISTIC-RULE PROGRAMS ON SPECIALIZATION SYSTEMS

Kiyoshi Akama[1] and Ekawit Nantajeewarawat[2]

[1]Information Initiative Center
Hokkaido University
Sapporo, Hokkaido 060-0811, Japan
akama@iic.hokudai.ac.jp

[2]Computer Science Program
Sirindhorn International Institute of Technology
Thammasat University
Pathumthani 12121, Thailand
ekawit@siit.tu.ac.th

Received August 2013; revised February 2014

Abstract. *In this paper, we formulate a class of deterministic programs, called deterministic-rule programs, on a specialization system, and formalize their procedural semantics, called the clause-model semantics, taking their recursive conditional state-transition structures into account. The proposed theory makes clear what it means for a deterministic-rule program to be correct with respect to a specification. It provides a basis for developing methods for synthesis of deterministic programs from declarative specifications. Taking a specialization system as a parameter, the theory is applicable to many concrete classes of deterministic-rule programs with various forms of data structures through parameter instantiation.*

**Keywords:** Deterministic computation, Specialization system, Program synthesis, Query-answering problem, Equivalent transformation

## 1. Introduction.

Program synthesis is concerned with the construction of a correct and efficient program from a given specification. A general class of specifications based on query-answering (QA) problems is considered in this paper. Given a specification, we want to construct a deterministic program that is correct with respect to it. Starting from an initial state constructed from an input query, a deterministic program uniquely determines a (finite or infinite) sequence of states, called a *computation*, and if the computation ends with a final state, an output value is obtained. A deterministic program is correct with respect to a specification iff for each query, the program yields a finite computation with an obtained output value being equal to the answer to the query defined by the specification.

We establish a class of deterministic programs that are suitable for program synthesis. A program in this class is called a *deterministic-rule program* (*D-rule program*). It is a sequence of *deterministic rules* (*D-rules*) for transformation of definite clauses, which are regarded as computation states. Application of D-rules is determined by pattern matching and their applicability condition, which together make them very expressive for computation control. We give a clear and elegant semantics, called the *clause-model semantics*, for D-rule programs, taking their recursive applicability-checking structures into consideration. Based on this semantics, the correctness of a D-rule program with respect to a specification is clearly defined. Since a specification can be represented using clauses, the clause-model semantics narrows the gaps between D-rule programs and specifications, thereby simplifying the difficulty of program synthesis.

The paper is organized as follows. After defining the class of specifications and that of program synthesis problems considered herein, Section 2 introduces the concept of a deterministic program in general and identifies the objective of the paper. Section 3 introduces D-rules by means of examples. Section 4 formulates D-rule programs on a specialization system. Section 5 describes computation states and computations of D-rule programs, and presents the clause-model semantics. Section 6 shows how D-rule programs are used for computing the answers to QA problems and illustrates program correctness and program synthesis on the space of D-rule programs. Section 7 summarizes the characteristics of the D-rule language and makes a comparison with Prolog [1], Guarded Horn Clause (GHC) [2], the nondeterministic-rule (N-rule) language [3], and imperative languages. Section 8 concludes the paper.

*Preliminary Notation.* The notation that follows holds thereafter. Given a set $A$, $pow(A)$ denotes the power set of $A$ and $partialMap(A)$ the set of all partial mappings on $A$ (i.e., from $A$ to $A$). For any sets $A$ and $B$, $f : A \nrightarrow B$ denotes a partial mapping $f$ from $A$ to $B$ and $partialMap(A, B)$ the set of all partial mappings from $A$ to $B$. For any partial mappings $f$ and $g$, $f \circ g$ denotes the composition of $f$ with $g$[1]. For any nonempty sequence $\mathbf{s}$, $first(\mathbf{s})$ denotes the first element of $\mathbf{s}$; $last(\mathbf{s})$ denotes the last element of $\mathbf{s}$ if $\mathbf{s}$ is finite; and $rest(\mathbf{s})$ denotes the sequence obtained from $\mathbf{s}$ by removing its first element. For any sequences $\mathbf{s}$ and $\mathbf{s}'$, if $\mathbf{s}$ is finite, $\mathbf{s} \cdot \mathbf{s}'$ denotes the concatenation of $\mathbf{s}$ and $\mathbf{s}'$.

## 2. Program Synthesis Problems and Design of Deterministic Programs.

First, the class of specifications and that of program synthesis problems considered in this paper are defined in Section 2.1. The concepts of a deterministic program, its computations, and its computed answers are given in Section 2.2. The objective of the paper is then identified in Section 2.3.

### 2.1. Specifications and program synthesis problems.

2.1.1. *QA problems on definite clauses.* A *query-answering problem* (*QA problem*) on definite clauses is a pair $\langle D, q\rangle$, where $D$ is a set of definite clauses, representing background knowledge, and $q$ is a user-defined atom, representing a query. The answer to a QA problem $\langle D, q\rangle$ is the set $\mathcal{M}(D) \cap rep(q)$, where $\mathcal{M}(D)$ is the least model of $D$ and $rep(q)$ is the set of all ground instances of $q$.

2.1.2. *Specifications.* A *specification* based on QA problems on definite clauses is a set of QA problems on definite clauses with common background knowledge, i.e., a set $\{\langle D, q\rangle \mid q \in Q\}$, where $D$ is a set of definite clauses and $Q$ is a set of query atoms. All specifications considered henceforth are specifications in this class.

A specification $\{\langle D, q\rangle \mid q \in Q\}$ is denoted by the pair $\langle D, Q\rangle$. Let $\mathcal{G}$ denote the set of all ground atoms. A specification $S = \langle D, Q\rangle$ determines a mapping $answers_S : Q \to pow(\mathcal{G})$ such that for any $q \in Q$, $answers_S(q) = \mathcal{M}(D) \cap rep(q)$.

2.1.3. *Program synthesis problems and program correctness.* A *program synthesis problem* is to find in a given program space a program that is correct with respect to a given specification. A program $P$ is *correct* with respect to a specification $S = \langle D, Q\rangle$ iff for any query atom $q \in Q$, (i) $P$ returns an answer to the QA problem $\langle D, q\rangle$ in finite time and (ii) the returned answer coincides with $answers_S(q)$.

[1] $f \circ g = \{\langle x, z\rangle \mid (\langle x, y\rangle \in g) \ \&\ (\langle y, z\rangle \in f)\}$.

2.2. **Deterministic programs and computations.** A deterministic program $P$ has the following characteristics:

1. $P$ is associated with a set $\textsc{Sta}$ of *states*, which includes a set $\textsc{Ini}$ of initial states and a set $\textsc{Fin}$ of final states.
2. $P$ is deterministic, i.e., for any given initial state $st \in \textsc{Ini}$, $P$ uniquely determines a (finite or infinite) sequence of states beginning with $st$, which is called a *computation* of $P$ from $st$.
3. A computation of $P$ contains at most one final state, which can occur only as its last state.
4. $P$ determines a partial mapping $comp_P : \textsc{Ini} \nrightarrow \textsc{Fin}$ as follows: let $st \in \textsc{Ini}$.
   (a) If the computation of $P$ from $st$ reaches a final state $st' \in \textsc{Fin}$ in finite steps, then $comp_P(st) = st'$.
   (b) If the computation of $P$ from $st$ ends in finite steps without reaching any final state in $\textsc{Fin}$, then $comp_P(st)$ is undefined.
   (c) If the computation of $P$ from $st$ does not end in finite steps, then $comp_P(st)$ is undefined.

A set $Q$ of query atoms is associated with a mapping $makeState_Q : Q \to \textsc{Ini}$ for determining initial states from query atoms. Associated with $\textsc{Fin}$ is a partial mapping $val : \textsc{Fin} \nrightarrow pow(\mathcal{G})$ for determining answers obtained from final states. Given a set $Q$ of query atoms, a deterministic program $P$ determines a partial mapping $compute(P, Q) : Q \nrightarrow pow(\mathcal{G})$ by

$$compute(P, Q) = val \circ comp_P \circ makeState_Q.$$

The obtained partial mapping $compute(P, Q)$ associates with query atoms in $Q$ the answers computed using $P$.

Given a specification $S = \langle D, Q \rangle$, a deterministic program $P$ is correct with respect to $S$ iff $compute(P, Q)(q) = answers_S(q)$ for any $q \in Q$.

2.3. **Objective.** The objective of this paper is to design a program space $\mathbb{P}$ that satisfies the following requirements:

1. Each program in the space $\mathbb{P}$ is deterministic.
2. The program space $\mathbb{P}$ has rich expressive power.
3. For any program $P$ in the space $\mathbb{P}$, the partial mapping $comp_P$ is clearly and elegantly defined.
4. The space $\mathbb{P}$ is suitable for program synthesis both practically and theoretically.

In this paper, we propose a program space $\mathbb{P}$ that fulfills these requirements and make the partial mapping $compute(P, Q)$ explicit for any program $P$ in $\mathbb{P}$ and any set $Q$ of query atoms.

## 3. D-Rules by Examples.

Next, D-rules are introduced by way of examples. For reasons of readability, D-rules in the domain of first-order terms are used.

3.1. **Built-in atoms, built-in evaluators, and user-defined atoms.** In a D-rule system, some atoms, including extralogical atoms, are used for specifying predefined operations, and are referred to as *built-in atoms*. Built-in atoms are evaluated by a predetermined *built-in evaluator*, and if the evaluation succeeds, it yields a substitution as a result. Examples of built-in atoms are equality atoms, i.e., atoms of the form $=(t, t')$, where $t$ and $t'$ are first-order terms. An equality atom represents a unification operation. If $t$ and $t'$ are unifiable, the evaluation of $=(t, t')$ succeeds, with the resulting substitution being their most general unifier.

Atoms other than *built-in atoms* are considered as *user-defined* atoms. User-defined atoms are target atoms for rule application; they are evaluated through transformation of definite clauses by applying D-rules. In the examples that follow, atoms of the forms $app(l, l', l'')$, $toSet(l, l')$, $member(t, l)$, and $occur(t, t')$ are user-defined atoms, where $l, l', l''$ are lists and $t, t'$ are terms. They are intended to mean, respectively, "appending the list $l'$ to the list $l$ yields the list $l''$", "$l'$ is the list representing the set obtained from the list $l$ by removing repeated elements", "the term $t$ is an element of the list $l$", and "the term $t$ occurs in the term $t'$".

3.2. **Simple D-rules.** Figure 1 shows two D-rules for rewriting *app*-atoms, where each variable begins with an asterisk. Their applicability is determined by pattern matching. As specified by its left side, the rule $r_1$ (respectively, $r_2$) matches any *app*-atom whose first argument is the empty list (respectively, a nonempty list). When applied, the rules $r_1$ and $r_2$ replace their target *app*-atoms with corresponding instances of atoms specified in their right sides. Table 1 illustrates transformation of definite clauses by application of these two rules and the built-in evaluator. The predicate *ans* stands for "answer" and the definite clause in the first row of the table is intended to mean "$*A$ is the answer if it is the result of appending the list [3] to the list [1,2]". When the first atom in the body of a definite clause is an *app*-atom, it is rewritten using either $r_1$ or $r_2$. When it is an equality atom, it is evaluated using the built-in evaluator. The label of the rule applied at each rule-application step is shown in the second column of the table. The letter '$e$' in the same column indicates a transformation step resulting from built-in atom evaluation. The transformation changes the initial clause into the unit clause $ans([1, 2, 3]) \leftarrow$, which means "the list $[1, 2, 3]$ is the answer unconditionally".

$r_1$: $app([\,], *Y, *Z) \longrightarrow =(*Y, *Z).$
$r_2$: $app([*a|*X], *Y, *Z) \longrightarrow =(*Z, [*a|*W]), app(*X, *Y, *W).$

Figure 1. D-rule examples

Table 1. An example of a transformation sequence using the rules in Figure 1

| Step | Rule | Definite clause |
|---|---|---|
| | | $ans(*A) \leftarrow app([1, 2], [3], *A)$ |
| 1 | $r_2$ | $ans(*A) \leftarrow =(*A, [1|*W1]), app([2], [3], *W1)$ |
| 2 | $e$ | $ans([1|*W1]) \leftarrow app([2], [3], *W1)$ |
| 3 | $r_2$ | $ans([1|*W1]) \leftarrow =(*W1, [2|*W2]), app([\,], [3], *W2)$ |
| 4 | $e$ | $ans([1, 2|*W2]) \leftarrow app([\,], [3], *W2)$ |
| 5 | $r_1$ | $ans([1, 2|*W2]) \leftarrow =([3], *W2)$ |
| 6 | $e$ | $ans([1, 2, 3]) \leftarrow$ |

$r_3$: $toSet([\,], *Z) \longrightarrow =(*Z, [\,]).$
$r_4$: $toSet([*a|*X], *Z), \{member(*a, *X)\} \longrightarrow toSet(*X, *Z).$
$r_5$: $toSet([*a|*X], *Z) \longrightarrow =(*Z, [*a|*W]), toSet(*X, *W).$
$r_6$: $member(*a, [*a|*X]) \longrightarrow .$
$r_7$: $member(*a, [*b|*X]) \longrightarrow member(*a, *X).$

Figure 2. D-rule examples

TABLE 2. An example of a transformation sequence using the rules in Figure 2

| Step | Rule | Definite clause |
|---|---|---|
| | | $ans(*A) \leftarrow toSet([1,2,1],*A)$ |
| 1 | $r_4$ | $ans(*A) \leftarrow toSet([2,1],*A)$ |
| 2 | $r_5$ | $ans(*A) \leftarrow =(*A,[2 \mid *W1]), toSet([1],*W1)$ |
| 3 | $e$ | $ans([2 \mid *W1]) \leftarrow toSet([1],*W1)$ |
| 4 | $r_5$ | $ans([2 \mid *W1]) \leftarrow =(*W1,[1 \mid *W2]), toSet([],*W2)$ |
| 5 | $e$ | $ans([2,1 \mid *W2]) \leftarrow toSet([],*W2)$ |
| 6 | $r_3$ | $ans([2,1 \mid *W2]) \leftarrow =(*W2,[])$ |
| 7 | $e$ | $ans([2,1]) \leftarrow []$ |

TABLE 3. Checking whether $r_4$ is applicable at the first step in Table 2

| Step | Rule | Definite clause |
|---|---|---|
| | | $toSet([2,1],*A) \leftarrow member(1,[2,1])$ |
| 1 | $r_7$ | $toSet([2,1],*A) \leftarrow member(1,[1])$ |
| 2 | $r_6$ | $toSet([2,1],*A) \leftarrow []$ |

$r_8$: $occur(*a,*a) \longrightarrow .$
$r_9$: $occur(*a,[*A \mid *B]), \{occur(*a,*A)\} \longrightarrow .$
$r_{10}$: $occur(*a,[*A \mid *B]) \longrightarrow occur(*a,*B).$

FIGURE 3. D-rule examples

3.3. **D-rules with applicability conditions.** In addition to pattern matching, some applicability condition may be specified in order to confine a rule to being applicable to a more specific class of target atoms, enabling more specific atom replacement. Figure 2 shows D-rules for rewriting *toSet*-atoms and *member*-atoms. The rule $r_4$ has an applicability condition, i.e., $\{member(*a,*X)\}$, which indicated using a pair of curly braces in its left side. When its target-atom pattern, i.e., $toSet([*a \mid *X],*Z)$, is instantiated into a body atom using a substitution $\theta$, its corresponding instantiated applicability condition is checked, i.e., $member(*a\theta,*X\theta)$ is evaluated. A separate transformation sequence is constructed for the evaluation. Consider, for example, the definite clause in the first row of Table 2. To check whether $r_4$ is applicable to the body atom $toSet([1,2,1],*A)$, the instantiated condition $member(1,[2,1])$ is evaluated using the transformation in Table 3. Since this transformation ends with a unit clause, indicating the success of the evaluation, $r_4$ is applied, resulting in the first transformation step in Table 2.

For each transformation step, a rule is selected deterministically: the applicabilities of rules are checked one-by-one in the rule appearance order and (only) the first applicable rule is selected. For example, to make the second transformation step in Table 2, the applicability of $r_4$ to the target atom $toSet([2,1],*A)$ is checked first, i.e., $member(2,[1])$ is evaluated. Since the evaluation fails, the next rule, i.e., $r_5$, which is applicable to the target atom, is used at that step.

3.4. **D-rules with recursive applicability conditions.** It is often natural to write a D-rule with a recursive applicability condition. Consider the D-rules for rewriting *occur*-atoms in Figure 3. The rule $r_9$ removes an atom of the form $occur(t_a,[t_A \mid t_B])$, where $t_a,t_A,t_B$ are terms, if $t_a$ occurs in $t_A$, i.e., if $occur(t_a,t_A)$ is satisfied. Table 4 illustrates a

TABLE 4. An example of a transformation sequence using the rules in Figure 3

| Step | Rule | Definite clause |
|---|---|---|
| | | $ans \leftarrow occur(1, [[2,1,0],[5,4,3]])$ |
| 1 | $r_9$ | $ans \leftarrow []$ |

TABLE 5. Checking whether $r_9$ is applicable at the first step in Table 4

| Step | Rule | Definite clause |
|---|---|---|
| | | $ans \leftarrow occur(1, [2,1,0])$ |
| 1 | $r_{10}$ | $ans \leftarrow occur(1, [1,0])$ |
| 2 | $r_9$ | $ans \leftarrow []$ |

TABLE 6. A transformation failure

| Step | Rule | Definite clause |
|---|---|---|
| | | $ans \leftarrow occur(1, 2)$ |
| 1 | – | $\bot$ |

transformation sequence using the rules in Figure 3[2]. To determine whether $r_9$ is applicable to the body atom $occur(1, [[2,1,0],[5,4,3]])$, the condition $occur(1, [2,1,0])$ is checked, initiating the transformation sequence in Table 5. To make the first transformation step in Table 5, the applicability of $r_9$ to $occur(1, [2,1,0])$ is determined, i.e., the condition $occur(1, 2)$ is checked recursively using the transformation sequence in Table 6. Since none of $r_8$, $r_9$, and $r_{10}$ is applicable to $occur(1, 2)$, the transformation in Table 6 fails, which is indicated by the symbol '$\bot$'. The rule $r_9$ is thus not applicable at the first step in Table 5 and the next rule, i.e., $r_{10}$, is used at that step.

3.5. **Passing values through applicability conditions.** Evaluating an applicability condition may yield a substitution that instantiates body atoms, having the effect of passing a value to them. To illustrate, let us assume that (1) a *calc*-atom is a two-argument built-in atom that takes its first argument as an input for performing some calculation and outputs the calculation result as its second argument, and (2) a *cond*-atom is a one-argument built-in atom that tests whether its argument satisfies a certain condition. Now consider the following two rules:

$$\begin{aligned}&calcCond(*x, *z), \{calc(*x, *y), cond(*y)\} \longrightarrow =(*z, *y).\\ &calcCond(*x, *z) \longrightarrow =(*z, *x).\end{aligned}$$

Suppose that a target atom $calcCond(t_x, *z)$ is given, where $t_x$ is a term representing an input. To transform this target atom, the applicability of the first rule to it is checked first, i.e., a value $t_y$ is produced from $t_x$ using a *calc*-atom and then the condition $cond(t_y)$ is evaluated. If the evaluation of $cond(t_y)$ succeeds, the first rule is applied, i.e., the target atom is replaced with the equality atom $=(*z, t_y)$. If the evaluation of $cond(t_y)$ fails, the second rule is used instead, by which the target atom is replaced with $=(*z, t_x)$.

Supposing that the first rule is applied, the value $t_y$ is in turn passed to $*z$ by the evaluation of the equality atom $=(*z, t_y)$. One may produce the same effect by changing the first rule into

$$calcCond(*x, *z), \{calc(*x, *y), cond(*y)\} \longrightarrow calc(*x, *z).$$

[2]When it takes no argument, an *ans*-atom means "true".

The application of this new rule does not pass the value $t_y$ to its right side. Instead, it replaces the target atom with $calc(t_x, *z)$, the evaluation of which assigns the value $t_y$ directly to $*z$. This alternative, however, requires two *calc*-atoms to be evaluated – one for rule applicability checking and another for instantiating $*z$ – with their input arguments being the same. If the evaluation of a *calc*-atom takes high cost, the new rule is apparently less efficient.

## 4. D-Rule Programs on Specialization Systems.

After recalling the notion of a specialization system in Section 4.1, we formulate on it a general class of D-rule programs in Section 4.2.

### 4.1. Specialization systems.

The concept of a specialization system, introduced in [4], provides an axiomatic structure for studying the common interrelations between various forms of extended atoms and specialization operations on them. It provides a basis for a discussion of data structure extension [5] and for formulating declarative descriptions in many data domains, e.g., typed feature terms [6], conceptual graphs [7], and XML expressions [8]. A specialization system is recalled below.

**Definition 4.1.** *A specialization system* $\Gamma$ *is a quadruple* $\langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ *of three sets* $\mathcal{A}$*,* $\mathcal{G}$*, and* $\mathcal{S}$*, and a mapping* $\mu$ *from* $\mathcal{S}$ *to partialMap*$(\mathcal{A})$ *that satisfies the following conditions:*

1. $(\forall s', s'' \in \mathcal{S})(\exists s \in \mathcal{S}) : \mu(s) = \mu(s') \circ \mu(s'')$.
2. $(\exists s \in \mathcal{S})(\forall a \in \mathcal{A}) : \mu(s)(a) = a$.
3. $\mathcal{G} \subseteq \mathcal{A}$.

*Elements of* $\mathcal{A}$*,* $\mathcal{G}$*, and* $\mathcal{S}$ *are called atoms, ground atoms, and specializations, respectively. The mapping* $\mu$ *is called the specialization operator of* $\Gamma$*. A specialization* $s \in \mathcal{S}$ *is said to be applicable to* $a \in \mathcal{A}$ *iff* $a \in dom(\mu(s))$*.* ■

In the rest of this paper, assume that a specialization system $\Gamma = \langle \mathcal{A}, \mathcal{G}, \mathcal{S}, \mu \rangle$ is given. A specialization in $\mathcal{S}$ is often denoted by a Greek letter such as $\theta$. A specialization $\theta \in \mathcal{S}$ is often identified with the partial mapping $\mu(\theta)$ and used as a postfix unary (partial) operator on $\mathcal{A}$, e.g., $\mu(\theta)(a) = a\theta$, provided that no confusion is caused. Given an expression $E$ containing occurrences of atoms in $\mathcal{A}$ and $\theta \in \mathcal{S}$ such that $\theta$ is applicable to all those atoms, we write $E\theta$ to denote the expression obtained from $E$ by replacing each atom $a \in \mathcal{A}$ that occurs in $E$ with $a\theta$. Let $\epsilon$ denote the identity specialization in $\mathcal{S}$, i.e., $a\epsilon = a$ for any $a \in \mathcal{A}$. For any $\theta, \sigma \in \mathcal{S}$, let $\theta \circ \sigma$ denote a specialization $\rho \in \mathcal{S}$ such that $\mu(\rho) = \mu(\sigma) \circ \mu(\theta)$, i.e., $a(\theta \circ \sigma) = (a\theta)\sigma$ for any $a \in \mathcal{A}$.

A subset $A'$ of $\mathcal{A}$ is said to be *closed* iff for any $a \in A'$ and any $\theta \in \mathcal{S}$, and if $\theta$ is applicable to $a$, then $a\theta \in A'$.

### 4.2. D-rule programs on specialization systems.

D-rules and D-rule programs on $\Gamma$ are defined below. Assume that $\mathcal{A}_\mathrm{B}$ and $\mathcal{A}_\mathrm{U}$ are disjoint closed subsets of $\mathcal{A}$ such that $\mathcal{A}_\mathrm{B} \cup \mathcal{A}_\mathrm{U} = \mathcal{A}$. Atoms in $\mathcal{A}_\mathrm{B}$ are regarded as built-in atoms, and those in $\mathcal{A}_\mathrm{U}$ as user-defined atoms.

A *deterministic rule* (for short, *D-rule*) $r$ on $\Gamma$ is an expression of the form

$$h, \{c_1, \ldots, c_m\} \longrightarrow b_1, \ldots, b_n,$$

where $h \in \mathcal{A}_\mathrm{U}$; $m, n \geq 0$; $c_i \in \mathcal{A}_\mathrm{B} \cup \mathcal{A}_\mathrm{U}$ for each $i \in \{1, \ldots, m\}$; and $b_j \in \mathcal{A}_\mathrm{B} \cup \mathcal{A}_\mathrm{U}$ for each $j \in \{1, \ldots, n\}$. The user-defined atom $h$ is called the *head* of $r$, denoted by *head*$(r)$. The sequences $[c_1, \ldots, c_m]$ and $[b_1, \ldots, b_n]$ are called the *applicability condition* of $r$, denoted by *cond*$(r)$, and the *body* of $r$, denoted by *body*$(r)$, respectively. The pair of braces on the left side of $r$ does not indicate the set notation; the order of atoms in the applicability condition as well as the order of those in the body of $r$ is important. When

$m = 0$, the pair of braces on the left side of $r$ is omitted. A specialization $\theta \in \mathcal{S}$ is said to be *applicable* to $r$ iff $\theta$ is applicable to each atom occurring in $r$. Let $\textsc{Drule}(\Gamma)$ denote the set of all D-rules on $\Gamma$.

A *deterministic-rule program* (for short, *D-rule program*) on $\Gamma$ is a finite sequence of D-rules on $\Gamma$.

## 5. Computations of a D-Rule Program and Clause-Model Semantics.

First, a computation and a complete computation set are defined in Section 5.1. Applicability of D-rules to states with respect to a complete computation set is then defined in Section 5.2. The conditional state-transition structure of a D-rule program is described in Section 5.3. It is followed by a formalization of the clause-model semantics in Section 5.4.

### 5.1. States and computations.

A *clause-model state* (for short, *C-state*) on $\Gamma$ takes one of the following forms:

1. $\langle (ans(as) \leftarrow gs), \rho \rangle$, where $as$ and $gs$ are finite sequences of atoms in $\mathcal{A}$ and $\rho$ is a specialization in $\mathcal{S}$.
2. A special symbol $\perp$, which is called the *null C-state.*

An *initial C-state* on $\Gamma$ is a C-state on $\Gamma$ of the first form such that $\rho$ is the identity specialization $\epsilon$. A *final C-state* on $\Gamma$ is either the null C-state ($\perp$) or a C-state on $\Gamma$ of the first form such that $gs$ is the empty sequence. Let $\textsc{Sta}$, $\textsc{Ini}$, and $\textsc{Fin}$ be the set of all C-states, the set of all initial C-states, and the set of all final C-states, respectively, on $\Gamma$. When no confusion is caused, a C-state is simply called a *state*.

**Definition 5.1.** *A computation on* $\textsc{Sta}$ *is a nonempty sequence of states in* $\textsc{Sta}$*. A computation on* $\textsc{Sta}$ *is said to be infinite if it is an infinite sequence, and it is said to be finite otherwise. A computation com on* $\textsc{Sta}$ *is said to be complete with respect to* $\textsc{Ini}$ *and* $\textsc{Fin}$ *iff it satisfies the following conditions:*

1. *first*$(com) \in \textsc{Ini}$.
2. *If com is finite, then last*$(com) \in \textsc{Fin}$. ■

**Definition 5.2.** *A set $M$ of computations on* $\textsc{Sta}$ *is deterministic iff for any* $com, com' \in M$*, if first*$(com) =$ *first*$(com')$*, then* $com = com'$. ■

**Definition 5.3.** *A set $M$ of computations on* $\textsc{Sta}$ *is complete with respect to* $\textsc{Ini}$ *and* $\textsc{Fin}$ *iff the following conditions are satisfied:*

1. *For any computation com in $M$, com is complete with respect to* $\textsc{Ini}$ *and* $\textsc{Fin}$.
2. *$M$ is deterministic.*

*Let* $\textsc{cmpComp}(\textsc{Sta}, \textsc{Ini}, \textsc{Fin})$ *denote the set of all complete sets of computations on* $\textsc{Sta}$ *with respect to* $\textsc{Ini}$ *and* $\textsc{Fin}$. ■

When no confusion is caused, $\textsc{cmpComp}(\textsc{Sta}, \textsc{Ini}, \textsc{Fin})$ is also written as $\textsc{cmpComp}$.

Let a partial mapping $fin : \textsc{cmpComp} \times \textsc{Ini} \nrightarrow \textsc{Fin}$ be defined as follows: for any $M \in \textsc{cmpComp}$ and any $st \in \textsc{Ini}$, if $M$ contains a finite computation $com$ such that $first(com) = st$ and $last(com) \in \textsc{Fin}$, then $fin(M, st) = last(com)$; otherwise it is undefined.

### 5.2. Applicability of D-rules with respect to a complete computation set.

To apply a rule to a state, a pattern-matching specialization is determined. If no pattern matching is possible, the rule is not applicable. The determination of pattern-matching specializations is specified by a mapping

$$\mathbf{m} : (\textsc{Drule}(\Gamma) \times (\textsc{Sta} - \textsc{Fin})) \to (\mathcal{S} \cup \{\perp\})$$

such that for any rule $r \in \textsc{Drule}(\Gamma)$ and any state $st = \langle (ans(as) \leftarrow gs), \rho \rangle \in \textsc{Sta} - \textsc{Fin}$, if $\mathbf{m}(r, st)$ is a specialization, say $\theta$, in $\mathcal{S}$, then

1. $\theta$ is applicable to $r$ (i.e., it is applicable to $head(r)$, $cond(r)$, and $body(r)$), and
2. $head(r)\theta = first(gs)$.

Intuitively, given a D-rule $r$ and a state $st = \langle (ans(as) \leftarrow gs), \rho \rangle \in \textsc{Sta} - \textsc{Fin}$,

- if $\mathbf{m}(r, st)$ is a specialization $\theta \in \mathcal{S}$, then $head(r)$ is matched with $first(gs)$ by $\theta$ and the applicability of $r$ to $st$ will be checked further by considering $cond(r)\theta$; and
- if $\mathbf{m}(r, st) = \bot$, then $r$ is not applied.

Now let $M$ be a complete set of computations and $st \in \textsc{Sta} - \textsc{Fin}$. Applicability and non-applicability of a D-rule to $st$ with respect to $M$ are defined as follows:

**Definition 5.4.** *A D-rule $r$ is applicable to $st$ with respect to $M$ iff $\mathbf{m}(r, st)$ is a specialization, say $\theta$, in $\mathcal{S}$ and $M$ contains a finite computation $com$ such that*

1. *$first(com) = \langle (ans(body(r)\theta) \leftarrow cond(r)\theta), \epsilon \rangle \in \textsc{Ini}$, and*
2. *$last(com) = \langle (ans(body(r)\theta\sigma) \leftarrow []), \sigma \rangle \in \textsc{Fin}$ for some specialization $\sigma \in \mathcal{S}$.*

*The specialization $\sigma$ in Condition 2 above is unique if it exists, and is referred to as $spec(r, st, M)$.* ■

**Definition 5.5.** *A D-rule $r$ is non-applicable to $st$ with respect to $M$ iff either (i) $\mathbf{m}(r, st) = \bot$ or (ii) $\mathbf{m}(r, st)$ is a specialization, say $\theta$, in $\mathcal{S}$ and $M$ contains a finite computation $com$ such that*

1. *$first(com) = \langle (ans(body(r)\theta) \leftarrow cond(r)\theta), \epsilon \rangle \in \textsc{Ini}$, and*
2. *$last(com) = \bot$.* ■

It follows directly that:

**Proposition 5.1.** *Let $r$ be a D-rule, $st \in \textsc{Sta} - \textsc{Fin}$, and $M, M' \in \textsc{cmpComp}$ such that $M \subseteq M'$. If $r$ is applicable to $st$ with respect to $M$, then $r$ is applicable to $st$ with respect to $M'$. If $r$ is non-applicable to $st$ with respect to $M$, then $r$ is non-applicable to $st$ with respect to $M'$.* ■

5.3. **Successor states with respect to a complete computation set.** Given a complete set $M$ of computations, the conditional state-transition structure of a D-rule program $P$ with respect to $M$ is characterized by a partial mapping $condSucc_P(M)$, which determines the successors of states by considering computations in $M$ for evaluating rule applicability conditions and for computing specializations used for constructing the successors. More precisely, a mapping

$$condSucc_P : \textsc{cmpComp} \rightarrow partialMap(\textsc{Sta} - \textsc{Fin}, \textsc{Sta}),$$

called the *conditional successor mapping* of $P$, is defined as follows: let $M \in \textsc{cmpComp}$ and $st = \langle (ans(as) \leftarrow gs), \rho \rangle \in \textsc{Sta} - \textsc{Fin}$. Then:

1. If $first(gs)$ is a built-in atom whose evaluation fails, then $condSucc_P(M)(st) = \bot$.
2. If $first(gs)$ is a built-in atom whose evaluation yields a specialization $\sigma$ and $\sigma$ is applicable to $as$ and $gs$, then
$$condSucc_P(M)(st) = \langle (ans(as\sigma) \leftarrow rest(gs)\sigma), \rho \circ \sigma \rangle.$$
3. If $first(gs)$ is a user-defined atom and every D-rule in $P$ is non-applicable to $st$ with respect to $M$, then $condSucc_P(M)(st) = \bot$.
4. If $first(gs)$ is a user-defined atom and there exists a D-rule $r$ in $P$ such that
   (a) $r$ is applicable to $st$ with respect to $M$, and
   (b) every D-rule that precedes $r$ in $P$ is non-applicable to $st$ with respect to $M$,

then

$$condSucc_P(M)(st) \;=\; \langle (ans(as\sigma) \leftarrow body(r)\theta\sigma \cdot rest(gs)\sigma), \rho \circ \sigma \rangle,$$

where $\theta = \mathbf{m}(r, st)$ and $\sigma = spec(r, st, M)$.

The conditional state-transition structure formalized by the mapping $condSucc_P$ is an extension of a simple state-transition structure discussed in [9], where each step in a state-transition sequence is determined without considering other state-transition sequences.

Since a successor state can be known only when all necessary computations required for checking relevant rule applicability are supplied by $M$, $condSucc_P(M)$ is in general not total.

**Proposition 5.2.** *The mapping $condSucc_P$ is monotonic, i.e., for any $M, M' \in$ cmpComp, if $M \subseteq M'$, then $condSucc_P(M) \subseteq condSucc_P(M')$.*

**Proof:** Let $M, M' \in$ cmpComp such that $M \subseteq M'$. Assume that $st \in$ Sta $-$ Fin, $st' \in$ Sta, and $(st, st') \in condSucc_P(M)$. It follows from Proposition 5.1 that $(st, st') \in condSucc_P(M')$. ■

5.4. **Clause-model semantics.** Based on the mapping $condSucc_P$, a mapping $K_P$ on cmpComp is associated with $P$ as follows:

**Definition 5.6.** *For any $M \in$ cmpComp, $K_P(M)$ is the set consisting of every computation com on* Sta *that satisfies the following conditions:*

1. *$first(com) \in$ Ini.*
2. *For any two successive states $st_i$ and $st_{i+1}$ in com, $condSucc_P(M)(st_i) = st_{i+1}$.*
3. *If com is finite, then $last(com) \in$ Fin.* ■

Obviously, $K_P(M)$ is deterministic for any $M \in$ cmpComp, and $K_P$ is a well-defined mapping from cmpComp to cmpComp. It is also readily seen that for any $M \in$ cmpComp, $K_P(M)$ contains a single-state computation $[st]$ for any final state $st \in$ Fin.

It follows directly from Proposition 5.2 that:

**Proposition 5.3.** *$K_P$ is monotonic.* ■

For any $n \geq 0$, let $K_P^n(\varnothing)$ be defined by: $K_P^0(\varnothing) = \varnothing$ and if $n \geq 1$, then $K_P^n(\varnothing) = K_P(K_P^{n-1}(\varnothing))$. Using $K_P$, the meaning of $P$ is defined below.

**Definition 5.7.** *The meaning of $P$, denoted by $\mathbb{M}(P)$, is defined as $\lim_{n\to\infty} K_P^n(\varnothing)$.* ■

Since $K_P$ is monotonic, $\mathbb{M}(P)$ is well defined.

In the following examples, let *BComp* denote the set consisting of every computation *com* on Sta such that (i) *first*(*com*) is a non-null state and (ii) the atom sequence on the right-hand side of the clause in *first*(*com*) consists only of built-in atoms in $\mathcal{A}_B$.

$$\begin{array}{l}
[\langle (ans([app([1,2],[3],*A)]) \leftarrow [app([1,2],[3],*A)]), \epsilon \rangle,\\
\;\langle (ans([app([1,2],[3],*A)]) \leftarrow [=(*A,[1|*W1]), app([2],[3],*W1)]), \epsilon \rangle,\\
\;\langle (ans([app([1,2],[3],[1|*W1])]) \leftarrow [app([2],[3],*W1)]), \{*A/[1|*W1]\} \rangle,\\
\;\langle (ans([app([1,2],[3],[1|*W1])]) \leftarrow [=(*W1,[2|*W2]), app([],[3],*W2)]), \{*A/[1|*W1]\} \rangle,\\
\;\langle (ans([app([1,2],[3],[1,2|*W2])]) \leftarrow [app([],[3],*W2)]), \{*A/[1,2|*W2], *W1/[2|*W2]\} \rangle,\\
\;\langle (ans([app([1,2],[3],[1,2|*W2])]) \leftarrow [=([3],*W2)]), \{*A/[1,2|*W2], *W1/[2|*W2]\} \rangle,\\
\;\langle (ans([app([1,2],[3],[1,2,3])]) \leftarrow []), \{*A/[1,2,3], *W1/[2,3], *W2/[3]\} \rangle]
\end{array}$$

Figure 4. An example of a computation

**Example 5.1.** *Referring to the D-rules $r_1$ and $r_2$ in Figure 1 (Section 3.2), let $P$ be the D-rule program $[r_1, r_2]$. Obviously, $K_P^1(\varnothing) = BComp \cup \{[st] \mid st \in \textsc{Fin}\}$. Figure 4 gives an example of a computation in $K_P^2(\varnothing) - K_P^1(\varnothing)$.* ■

**Example 5.2.** *Next, consider the D-rules $r_8$, $r_9$, and $r_{10}$ in Figure 3 (Section 3.4). Let $P$ be the D-rule program $[r_8, r_9, r_{10}]$. Again $K_P^1(\varnothing) = BComp \cup \{[st] \mid st \in \textsc{Fin}\}$. Consider the sets $K_P^2(\varnothing) - K_P^1(\varnothing)$ and $K_P^3(\varnothing) - K_P^1(\varnothing)$. The former set is a proper subset of the latter one. Both of them contain, for example, the following computations:*

1. $[\langle (ans([occur(3,2)]) \leftarrow [occur(3,2)]), \epsilon\rangle, \bot]$
2. $[\langle (ans([occur(3,3)]) \leftarrow [occur(3,3)]), \epsilon\rangle, \langle (ans([occur(3,3)]) \leftarrow []), \epsilon\rangle]$

*Figure 5 illustrates a computation that belongs to the latter set but not the former one.* ■

$$\begin{array}{l}[\langle (ans([occur(3,[2,3])]) \leftarrow [occur(3,[2,3])]), \epsilon\rangle, \\ \langle (ans([occur(3,[2,3])]) \leftarrow [occur(3,[3])]), \epsilon\rangle, \\ \langle (ans([occur(3,[2,3])]) \leftarrow []), \epsilon\rangle]\end{array}$$

FIGURE 5. An example of a computation

## 6. Query Answering and Program Synthesis on the Space of D-Rule Programs.

Following the general scheme established in Section 2.2, how the clause-model semantics characterizes D-rule programs in the context of query answering is described in Section 6.1. Along with program correctness, program synthesis on the space of D-rule programs is illustrated in Section 6.2.

### 6.1. Query answering using D-rule programs.

Let $P$ be a D-rule program. Referring to the general scheme in Section 2.2 and the partial mapping *fin* in Section 5.1, how $\mathbb{M}(P)$ determines the computed answers to query atoms is given below. Assume that $Q$ is a set of query atoms. First, a mapping $makeState_Q : Q \to \textsc{Ini}$, a partial mapping $comp_P : \textsc{Ini} \nrightarrow \textsc{Fin}$, and a partial mapping $val : \textsc{Fin} \nrightarrow pow(\mathcal{G})$ are given as follows:

- For any query atom $q \in Q$, $makeState_Q(q)$ is the initial state $\langle (ans([q]) \leftarrow [q]), \epsilon\rangle$.
- For any initial state $st \in \textsc{Ini}$, $comp_P(st) = fin(\mathbb{M}(P), st)$.
- $val(\bot) = \varnothing$ and for any non-null final state $st = \langle (ans(as) \leftarrow []), \rho\rangle \in \textsc{Fin}$, if $as$ is a singleton sequence $[a]$, then $val(st) = rep(a)$; otherwise $val(st)$ is undefined.

Next, the computed answers to query atoms in $Q$ are determined by the partial mapping $compute(P,Q) : Q \nrightarrow pow(\mathcal{G})$ defined by $compute(P,Q) = val \circ comp_P \circ makeState_Q$, i.e.,

$$compute(P,Q)(q) = val(fin(\mathbb{M}(P), makeState_Q(q)))$$

for any $q \in Q$. It is readily seen that for any $q \in Q$, $compute(P,Q)(q)$ can be restated as follows:

1. $compute(P,Q)(q) = rep(q')$ if $\mathbb{M}(P)$ contains a finite computation *com* such that $first(com) = makeState_Q(q)$ and $last(com) = \langle (ans([q']) \leftarrow []), \rho\rangle$ for some specialization $\rho \in \mathcal{S}$.
2. $compute(P,Q)(q) = \varnothing$ if $\mathbb{M}(P)$ contains a finite computation *com* such that $first(com) = makeState_Q(q)$ and $last(com) = \bot$.
3. $compute(P,Q)(q)$ is undefined otherwise.

Given a query atom $q \in Q$, $compute(P,Q)(q)$ is the *computed answer* to $q$ obtained using the D-rule program $P$.

6.2. **Program synthesis problems for query answering.** A D-rule program $P$ is *correct* with respect to a specification $S = \langle D, Q\rangle$ iff for any $q \in Q$, $\mathit{compute}(P, Q)(q) = \mathit{answers}_S(q)$. Let $\mathbb{P}$ be the set of all D-rule programs. A *program synthesis problem* for a specification $S$ on the space $\mathbb{P}$ is to construct a D-rule program that is correct with respect to $S$.

**Example 6.1.** *Assume that $S$ is a specification $\langle D, Q\rangle$, where $D$ consists of the two definite clauses*

1. $\mathit{app}([], *Y, *Y) \leftarrow$,
2. $\mathit{app}([*a|*X], *Y, [*a|*Z]) \leftarrow \mathit{app}(*X, *Y, *Z)$,

*which together provide the definition of the predicate app, and*

$$Q = \{\mathit{app}(l_1, l_2, v) \mid (l_1 \textit{ and } l_2 \textit{ are ground lists}) \;\&\; (v \textit{ is a variable})\}.$$

*An atom $\mathit{app}(l_1, l_2, v)$ in $Q$ represents the query "find the result of appending $l_2$ to $l_1$" and the answer to this query is given by instantiating $v$.*

*Let $P$ be the D-rule program $[r_1, r_2]$, where $r_1$ and $r_2$ are the D-rules in Figure 1 (Section 3.2). $P$ is correct with respect to the specification $S$, the reasons being as follows:*

- *From the background knowledge $D$, the following equivalent transformation (ET) rules* [3] *are derived:*
  1. *An atom of the form $\mathit{app}([], *Y, *Z)$ can be replaced equivalently with a built-in equality atom of the form $=(*Y, *Z)$.*
  2. *An atom of the form $\mathit{app}([*a|*X], *Y, *Z)$ can be replaced equivalently with an atom set of the form $\{=(*Z, [*a|*W]), \mathit{app}(*X, *Y, *W)\}$.*

  *Since $r_1$ and $r_2$ are directly obtained, respectively, from these two ET rules, each state-transition step of $P$ is always an equivalent transformation step with respect to $D$ (see, e.g., Figure 4). Hence for any $q \in Q$, if the computation of $P$ for $q$ ends with a final state, then the obtained computed answer always coincides with the answer to $q$ defined by the specification $S$.*
- *Let com be any arbitrary computation of $P$ that begins with $\mathit{makeState}_Q(q)$ for some $q \in Q$. It is readily seen that com possesses the following properties:*
  - *The right-hand side of the clause in each non-final state in com comprises at most one equality atom and at most one app-atom, which has ground lists as its first and second arguments. An equality atom is always processable by the built-in evaluator. One of $r_1$ and $r_2$ is always applicable to any app-atom in a non-final state in com. Therefore, $P$ always makes state transition on any non-final state possibly occurring in com.*
  - *When $r_2$ is applied to an app-atom whose first argument is a ground nonempty list, the length of that first argument list decreases by one. As a result, after a finite number of application of $r_2$, $r_1$ is always applicable. Application of $r_1$ followed by the evaluation of an equality atom always yields a final state. So com terminates within finite state-transition steps.*

  *Hence for any $q \in Q$, the computation of $P$ for $q$ always terminates and reaches a final state in finite time.* ■

7. **Comparison with Related Rule Languages.** Table 7 compares D-rule programs with Prolog [1], Guarded Horn Clause (GHC) [2], and nondeterministic-rule (N-rule) programs [3] from the following viewpoints:

(Ch1) Single-clause state
(Ch2) Deterministic computation
(Ch3) Recursive applicability condition

(Ch4) Instantiation into specific classes of programs through specialization systems
(Ch5) Appropriateness for synthesis of deterministic programs

Characteristics of D-rule programs from these viewpoints and the comparison in Table 7 are described below.

TABLE 7. Comparing D-rule, Prolog, GHC, and N-rule programs

| Language/ characteristic | (Ch1) | (Ch2) | (Ch3) | (Ch4) | (Ch5) |
|---|---|---|---|---|---|
| D-rule | ○ | ○ | ○ | ○ | ○ |
| Prolog | △ | ○ | △ | × | △ |
| GHC | ○ | × | △ | × | △ |
| N-rule | × | × | × | ○ | ○ |

○ – "good"; △ – "moderate"; × – "poor"

7.1. **Characteristics of D-rule programs.** A D-rule program is a sequence of D-rules defined on a specialization system. A computation state for a D-rule program is a single definite clause on a specialization system and a computation is a sequence of states obtained by successive application of D-rules. Computation of a D-rule program is deterministic, since the first atom in the right-hand side of a state is selected to be transformed by the first applicable D-rule in the program. D-rules have recursive applicability conditions. D-rule programs are appropriate for generation of deterministic programs for solving QA problems, since many useful D-rules can be constructed directly from equivalent transformation (ET) rules [3], which are derivable from clauses [10].

7.2. **Comparison.** A pure Prolog program [11] is a sequence of ordered definite clauses[3], which are conventionally regarded as rules for making backward chaining inference. The main structure of the procedural semantics of a Prolog program is a deterministic depth-first search for finding all answers to a given query. Such a search can be seen as a process of successively transforming a set of definite clauses using general unfolding-based rules [10]. Moreover, the applicability of a rule (an ordered definite clause) in pure Prolog is determined solely by unification of its head and a target atom; no applicability condition other than unification checking can be specified, resulting in the lack of expressive power for computation control. The extralogical predicate *cut* is introduced in Prolog as a remedy [1]. The sequence of atoms that precedes *cut* in the body of a rule can be considered as the applicability condition of the rule. However, the introduction of *cut* increases difficulty in semantics formalization, which is necessary for a discussion of program correctness and program synthesis.

Guarded Horn Clause (GHC) rules [2] yield single-clause transformation and appear to be more similar to D-rules than Prolog rules in regard to rule applicability checking. The applicability of a GHC rule is determined by pattern matching and GHC provides commit operators for specifying additional applicability conditions. However, the computation model of GHC is nondeterministic and parallel, while that of D-rule programs is deterministic and sequential. A simplified version of GHC, i.e., Flat GHC, is often used to reduce the cost of computing recursive applicability conditions. Research on GHC focusses more towards parallelism and efficiency, rather than program synthesis.

[3]An ordered definite clause is a definite clause whose right-hand side is regarded as an atom sequence, rather than an atom set.

A nondeterministic-rule (N-rule) program [3] is a set of nondeterministic rules (N-rules), which have no recursive applicability condition. A computation state for an N-rule program is a set of definite clauses on a specialization system and a computation is a sequence of states obtained by successive application of N-rules. Computation by an N-rule program is nondeterministic, i.e., a target definite clause in a computation state, a target atom for rule application, and an applicable N-rule is selected nondeterministically for making state transition. With regard to program synthesis, if an N-rule program comprises only ET rules, then its computation is guaranteed to yield a correct answer whenever a final state is reached.

Programs in most imperative languages are deterministic programs. However, the procedural semantics of existing imperative languages has little in common with specifications based on QA problems. It is thus difficult to establish relationships between programs in these languages and specifications in this class, and consequently, difficult to develop a theory for program synthesis based on these languages. Since the background knowledge of a specification comprises clauses, the use of clauses as computation states facilitates insightful understanding of the relationships between D-rule programs and specifications, and simplifies the difficulty of program synthesis. For example, as illustrated in Section 6.2, from a set of definite clauses representing background knowledge, ET rules [3] for equivalent transformation of clauses can be derived and they provide high-level intermediate artifacts for constructing a D-rule program. A method for systematically generating ET rules from a set of definite clauses, called the squeeze method, was presented in [10].

## 8. Conclusions.

In this paper, we have formulated a class of deterministic programs, called deterministic-rule (D-rule) programs. D-rule programs have been extensively used as part of an ET-based language called ETI. Many applications and experimental systems have been constructed using the ETI language. A D-rule program is similar to a nondeterministic-rule (N-rule) program in that (i) they are both defined with specialization parameters, (ii) they employ clauses as components of their computation states, thereby narrowing the gaps between them and declarative descriptions of QA problems, and (iii) they can be generated based on ET rules. Compared to nondeterministic computation by N-rule programs, D-rule programs can represent deterministic computation precisely and provide richer expressive power through their recursive conditional state-transition structures. Taking these features into account, we have formalized a procedural semantics of D-rule programs, called the clause-model semantics. This semantics is the only existing semantics that can clearly determine the correctness of a D-rule program with respect to a specification. It provides a basis for developing practical methods for synthesis of many deterministic programs from declarative specifications.

**Acknowledgment.** This research was supported by the Collaborative Research Programs 2012 and 2013, Information Initiative Center, Hokkaido University, and by the National Research University Project of Thailand Office of Higher Education Commission.

## REFERENCES

[1] L. Sterling and E. Shapiro, *The Art of Prolog*, MIT Press, 1986.
[2] K. Ueda, *Guarded Horn Clauses*, Ph.D. Thesis, University of Tokyo, Japan, 1986.
[3] K. Akama and E. Nantajeewarawat, Formalization of the equivalent transformation computation model, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol.10, no.3, pp.245-259, 2006.
[4] K. Akama, Declarative semantics of logic programs on parameterized representation systems, *Advances in Software Science and Technology*, vol.5, pp.45-63, 1993.

[5] K. Akama, H. Koike and H. Mabuchi, Equivalent transformation by safe extension of data structures, *Lecture Notes in Computer Science*, vol.2244, pp.140-148, 2001.

[6] E. Nantajeewarawat and V. Wuwongse, Defeasible inheritance through specialization, *Computational Intelligence*, vol.17, no.1, pp.62-86, 2001.

[7] V. Wuwongse and E. Nantajeewarawat, Declarative programs with implicit implication, *IEEE Transactions on Knowledge and Data Engineering*, vol.14, no.4, pp.836-849, 2002.

[8] V. Wuwongse et al., A data model for XML databases, *Journal of Intelligent Information Systems*, vol.20, no.1, pp.63-80, 2003.

[9] K. Akama and E. Nantajeewarawat, State-transition computation models and program correctness thereon, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol.11, no.10, pp.1250-1261, 2007.

[10] K. Akama, E. Nantajeewarawat and H. Koike, Program generation in the equivalent transformation computation model using the squeeze method, *Lecture Notes in Computer Science*, vol.4378, pp.41-54, 2007.

[11] J. W. Lloyd, *Foundations of Logic Programming*, 2nd Edition, Springer-Verlag, 1987.