State-Transition Computation Models and Program Correctness Thereon

Kiyoshi Akama
Information Initiative Center, Hokkaido University
Sapporo, Hokkaido, 060-0811, Japan
Email: akama@iic.hokudai.ac.jp

Ekawit Nantajeewarawat
Sirindhorn International Institute of Technology
Thammasat University, P.O. Box 22, Thammasat-Rangsit
Pathumthani 12121, Thailand
Email: ekawit@siit.tu.ac.th

Abstract—A common framework for formalization of state-transition computation models is presented based on a general theory for studying the interrelationships between specifications, programs, computations, and program correctness. A necessary and sufficient condition for program correctness on this class of computation models is established. Application of the framework is demonstrated by formalizing as its instances two concrete examples of state-transition computation models, called the NAT computation model and the D-RULE computation model. A comparison between these two computation models as regards their correct-program spaces is illustrated by introduction of the concept of an embedding mapping.

I. INTRODUCTION

Many computation models and programming languages exist in the literature. Comparisons between them are often difficult, owing not only to the differences in their syntax and semantics, but also to the lack of general common agreement on what properties should be considered and what can be neglected. When a new computation model is proposed, it is often hard to justify whether the model offers real advantages over existing ones from a certain important viewpoint. Clear identification of a viewpoint of interest and selection of an appropriate common framework for examining computation models from that viewpoint would remedy the situation. We are interested in a comparative study of computation models from the viewpoint of program synthesis.

Program correctness is essential to discussion of program synthesis. Theoretically, program synthesis can be viewed as a search in a certain program space for a sufficiently efficient program that is correct with respect to a given specification. In [2], we developed a general common framework, called a program synthesis framework, for studying the interrelationships between specifications, programs, computations, and program correctness. To the best of our knowledge, no other such framework is available in the literature. A program synthesis framework consists of two self-contained components, called a specification universe and a program universe, which altogether provide a structure for defining the notion of program correctness, i.e., “what it means for a program to be correct with respect to a specification”. Emphasis on program correctness, in addition to computation, makes the framework remarkably different from other computation theories, most of which focus attention on computation and computable functions.

Programs and specifications are considered in the framework of [2] as abstract entities characterized solely by axioms capturing their roles as regards program correctness. Such an axiomatic approach allows one to concentrate on concepts that are truly pertinent to the essence of program correctness, without being distracted by differences between various specific computation models and irrelevant details thereof. The approach consequently results in the generality and the wide applicability of the framework: many computation models, e.g., logic programming [9], constraint logic programming [5], [8], functional programming [7], and equivalent transformation [1], can be formalized as instances of it.

A. State-Transition Computation Models

This paper focuses on an important class of computation models, called state-transition computation models. It specializes the framework of [2] in the context of this class, and provides a necessary and sufficient condition for program correctness in that context. Reasons for giving special attention to this class of computation models include:

- **Wide applicability:** State transition is a common form of computation, and every sequential algorithm can be step-for-step simulated by an appropriate abstract state machine (Gurevich’s sequential abstract state machine thesis) [6]. Most computation models in the literature can be regarded as state-transition computation models.
- **Sharper characterization of programs:** In the framework of [2], a program is characterized by a set of computations that it possibly produces, but how the program determines these computations is not discussed as part of the framework. In a state-transition computation model, a program can be identified with a relation on states, representing all possible state transition that it can cause. This sharper characterization enables more insightful discussion of common interrelationships between programs and computations.
- **Differences from other conceptual frameworks:** Existing frameworks for abstract state machines, e.g. [3], provide no theory of program correctness. Besides, although they discuss state transition extensively, they are not aimed at formalizing and comparing computation models.
B. Objectives of the Paper

The main purpose of this paper is threefold:

1) Formalization of state-transition program universes:
The notion of a state-transition program universe will be formulated. The primary requirement is to provide a precise common structure for characterizing programs in terms of state transition that they possibly yield, and for specifying how such programs determine computations on given problems and how to obtain answers from computations.

2) Establishment of a necessary and sufficient condition for program correctness: The class of state-transition computation models receives from the basic framework of [2] the definition of program correctness. A necessary and sufficient condition for this concept will be established based on characteristic features of this class (Theorem 1).

3) Illustration of formalizing and comparing computation models: To demonstrate its application, two concrete examples of state-transition computation models, called the NAT computation model and the D-RULE computation model, are formalized using the presented framework. A comparative study of computation models based on this framework will be illustrated: once they are formalized using the same basis, the NAT computation model can be rigorously embedded in the D-RULE computation model, i.e., elements of the former model can be mapped into those of the latter one in such a way that the structure of the former model is preserved. It will be shown that such an embedding mapping always preserves the correctness of programs in the former model (Theorem 2). Consequently, its existence implies that the D-RULE computation model has a larger correct-program space, and thus, provides a better structure for searching for correct and efficient programs.

C. An Overview of the Remainder of the Paper

After reviewing the basic framework of [2] in Section II, the concept of program universe is specialized into that of state-transition program universe in Section III. A necessary and sufficient condition for program correctness on a state-transition computation model is presented in Section IV. Several examples of state-transition programs are given in Section V. Formalization of two illustrative concrete computation models, i.e., the NAT computation model and the D-RULE computation model, is shown in Sections VI and VII, respectively. Embedding the NAT computation model in the D-RULE computation model is discussed in Section VIII. Section IX provides concluding remarks.

D. Preliminary Notations

The following notations will be used. Given a set $A$, $\text{pow}(A)$ denotes the power set of $A$ and the identity relation on $A$ is the set $\{ (a, a) \mid a \in A \}$. For any binary relation $r$, $\text{dom}(r)$ denotes the domain of $r$. A partial mapping $f$ and a total mapping $g$ from a set $A$ to a set $B$ are denoted by $f : A \rightarrow B$ and $g : A \rightarrow B$, respectively. Given $f : A \rightarrow B$ and $a \in A$, $f(a)$ is said to be defined if $a \in \text{dom}(f)$, and is said to be undefined otherwise.

For any sequence $s$, $\text{len}(s)$ denotes the length of $s$; $\text{first}(s)$ denotes the first element of $s$ if $s$ is not empty; and $\text{last}(s)$ denotes the last element of $s$ if $s$ is finite and not empty. For any sequences $s$ and $s'$, if $s$ is finite, $s \cdot s'$ denotes the concatenation of $s$ and $s'$. Given a set $A$, $f_{\text{seq}}(A)$ denotes the set of all finite sequences of elements of $A$.

II. A Basic Framework for Formalization

To start with, the definitions of a specification universe, a program universe, and program correctness are recalled.

A. Specification Universes and Program Universes

A specification universe provides a structure for characterizing specifications in terms of the problems they cover, and for defining the answers to these problems. It is formulated as follows.

Definition 1: A specification universe $\Sigma$ is a 5-tuple $\langle \text{SPEC}, \text{PROB}, \text{ANS}, \text{cover}, \text{answer} \rangle$, where SPEC, PROB, and ANS are sets, and

1) $\text{cover} : \text{SPEC} \rightarrow \text{pow}(\text{PROB})$,
2) $\text{answer} : \text{SPEC} \times \text{PROB} \rightarrow \text{ANS}$.

The sets SPEC, PROB, and ANS are called the specification space, the problem space, and the answer space, respectively, of $\Sigma$. Their elements are called specifications, problems, and answers, respectively. Given a specification $S \in \text{SPEC}$, each problem in $\text{cover}(S)$ is said to be covered by $S$. Given a specification $S \in \text{SPEC}$ and a problem $\text{prb} \in \text{cover}(S)$, if $\text{answer}(S, \text{prb})$ is defined, it is called the answer to $\text{prb}$ with respect to $S$.

Since an answer space can be a power set, the answer to a problem is possibly an answer set (for example, see Subsection VII-B). Next, the notion of a program universe is recalled. It specifies a domain of programs, determines all possible computations of each program, and defines the answer obtained from each computation.

Definition 2: A program universe $\Pi$ is a 6-tuple $\langle \text{PROG}, \text{PROB}, \text{COMP}, \text{ANS}, \text{compute}, \text{obtain} \rangle$, where PROG, PROB, COMP, and ANS are sets, and

1) $\text{compute} : \text{PROG} \times \text{PROB} \rightarrow \text{pow}(\text{COMP})$,
2) $\text{obtain} : \text{PROG} \times \text{COMP} \rightarrow \text{ANS}$.

The sets PROG, PROB, COMP, and ANS are called the program space, the problem space, the computation space, and the answer space, respectively, of $\Pi$. Their elements are called programs, problems, computations, and answers, respectively. Given a program $\text{prg} \in \text{PROG}$ and a program $\text{prb} \in \text{PROB}$, if $\text{compute}(\text{prg}, \text{prb})$ is defined, then each computation in $\text{compute}(\text{prg}, \text{prb})$ is called a computation of $\text{prg}$ on $\text{prb}$. A program $\text{prg} \in \text{PROG}$ is said to be deterministic if for any $\text{prb} \in \text{PROB}$, there is at most one computation of $\text{prg}$ on $\text{prb}$, and it is said to be nondeterministic otherwise. Given a program $\text{prg} \in \text{PROG}$ and a computation $\text{com} \in \text{COMP}$, if $\text{obtain}(\text{prg}, \text{com})$ is defined, it is called the answer obtained from $\text{com}$ with respect to $\text{prg}$.
B. Program Synthesis Frameworks and Program Correctness

A specification universe \( \Sigma \) and a program universe \( \Pi \) are said to be compatible iff they have the same problem space and the same answer space. A program synthesis framework is a pair \( (\Sigma, \Pi) \) of a specification universe \( \Sigma \) and a program universe \( \Pi \) such that \( \Sigma \) and \( \Pi \) are compatible.

The concept of program correctness—“what it means for a program to be correct with respect to a specification”—will now be formulated. Let \( (\Sigma, \Pi) \) be a program synthesis framework, where \( \Sigma = \langle \text{SPEC, PROB, ANS, cover, answer} \rangle \) and \( \Pi = \langle \text{PROG, PROB, COMP, ANS, compute, obtain} \rangle \).

Definition 3: A program \( \text{prg} \in \text{PROG} \) is said to be correct with respect to a specification \( S \in \text{SPEC} \) on \( (\Sigma, \Pi) \) iff for each problem \( \text{prb} \in \text{cover}(S) \), the following conditions are satisfied:

1) \( \text{compute}(\text{prg}, \text{prb}) \) is defined.
2) \( \text{compute}(\text{prg}, \text{prb}) \neq \emptyset \).
3) For each \( \text{com} \in \text{compute}(\text{prg}, \text{prb}) \),
   a) \( \text{obtain}(\text{prg}, \text{com}) \) is defined, and
   b) if \( \text{answer}(S, \text{prb}) \) is defined, then \( \text{obtain}(\text{prg}, \text{com}) = \text{answer}(S, \text{prb}) \).

When \( (\Sigma, \Pi) \) is clear from the context, the qualification “on \( (\Sigma, \Pi) \)” is often dropped.

A necessary condition for \( \text{prg} \in \text{PROG} \) to be correct with respect to \( S \in \text{SPEC} \) is that for any \( \text{prb} \in \text{cover}(S) \), \( \text{prg} \) must yield at least one computation on \( \text{prb} \) and, moreover, each computation \( \text{com} \) of \( \text{prg} \) on \( \text{prb} \) must yield an answer; however, if \( \text{answer}(S, \text{prb}) \) is undefined, then the answer obtained from \( \text{com} \) can be any arbitrary element of the answer space \( \text{ANS} \).

Also note that if \( \text{prg} \in \text{PROG} \) is nondeterministic and is correct with respect to \( S \in \text{SPEC} \), then for any \( \text{prb} \in \text{cover}(S) \) such that \( \text{answer}(S, \text{prb}) \) is defined, the answers obtained from any two different computations of \( \text{prg} \) on \( \text{prb} \) necessarily coincide, i.e., the choice of computations of \( \text{prg} \) on \( \text{prb} \) does not affect the obtained answer.

III. State-Transition Program Universes

Computing in many models takes a certain form of state transition [6], and the class of state-transition program universes deserves special attention. A program universe in this class can be formalized as follows.

Definition 4: A state-transition program universe \( \Pi_{ST} \) is a 9-tuple

\[
\langle \text{PROG, PROB, STATE, INI, FIN, ANS, } \tau, \text{makeState, makeAns} \rangle
\]

where \( \text{PROG, PROB, STATE, and ANS} \) are sets, and
1) \( \text{INI} \subseteq \text{STATE} \),
2) \( \text{FIN} \subseteq \text{STATE} \),
3) \( \tau : \text{PROG} \rightarrow \text{pow}(\text{STATE} \setminus \text{FIN}) \times \text{STATE} \),
4) \( \text{makeState} : \text{PROG} \rightarrow \text{INI} \).

1) \( \text{pow}(\text{STATE} \setminus \text{FIN}) \times \text{STATE} \) is the set of all binary relations from \( \text{STATE} \setminus \text{FIN} \) to \( \text{STATE} \).

5) \( \text{makeAns} : \text{FIN} \rightarrow \text{ANS} \).

The sets \( \text{PROG, PROB, STATE, and ANS} \) are called the program space, the problem space, the state space, and the answer space, respectively, of \( \Pi_{ST} \). Their elements are called programs, problems, states, and answers, respectively. States in \( \text{FIN} \) are called final states.\(^2\) Given a program \( \text{prg} \in \text{PROG} \), \( \tau(\text{prg}) \) is called the next-state relation of \( \text{prg} \).

A program \( \text{prg} \in \text{PROG} \) is characterized in \( \Pi_{ST} \) solely by the next-state relation \( \tau(\text{prg}) \). States in \( \text{INI} \) are possible initial states. Given a problem \( \text{prb} \in \text{PROB} \), an initial state is determined using the mapping \( \text{makeState} \) and a computation of \( \text{prg} \) on \( \text{prb} \) is constructed by making state transition using \( \tau(\text{prg}) \) successively until no further transition is possible. Provided that the resulting computation is finite and its last state is a final state, the obtained answer is then determined from the last state using the mapping \( \text{makeAns} \).

More precisely, \( \Pi_{ST} \) determines a program universe

\[
\langle \text{PROG, PROB, COMP, ANS, compute, obtain} \rangle,
\]

where \( \text{COMP, compute, and obtain} \) are defined as follows:

1) \( \text{COMP} \) is the set of all nonempty sequences of states in \( \text{STATE} \).
2) For any \( \text{prg} \in \text{PROG} \) and \( \text{prb} \in \text{PROB} \), a computation of \( \text{prg} \) on \( \text{prb} \) is a nonempty finite or infinite sequence \( \text{com} = [\text{st}_0, \text{st}_1, \text{st}_2, \ldots] \) of states such that
   \( \text{st}_0 = \text{makeState}(\text{prb}) \),
   for any two successive states \( \text{st}_i, \text{st}_{i+1} \) in \( \text{com} \), \( \langle \text{st}_i, \text{st}_{i+1} \rangle \in \tau(\text{prg}) \),
   if \( \text{com} \) is finite, then \( \text{last}(\text{com}) \notin \text{dom}(\tau(\text{prg})) \), and \( \text{compute}(\text{prg}, \text{prb}) \) is the set of all computations of \( \text{prg} \) on \( \text{prb} \).
3) For any \( \text{prg} \in \text{PROG} \) and \( \text{com} \in \text{COMP} \), if \( \text{com} \) is finite and \( \text{last}(\text{com}) \in \text{FIN} \), then
   \[ \text{obtain}(\text{prg}, \text{com}) = \text{makeAns}(\text{last}(\text{com})); \]
   otherwise \( \text{obtain}(\text{prg}, \text{com}) \) is undefined.\(^4\)

Accordingly, a state-transition program universe is regarded as a program universe.

Note that since programs are characterized as binary relations, rather than functions, on states, construction of a computation of a program is in general nondeterministic.

IV. A Necessary and Sufficient Condition for Program Correctness

State-transition computation models receive from Definition 3 (Subsection II-B) the notion of program correctness. This section establishes a necessary and sufficient condition for program correctness for this class of computation models (Theorem 1) based on the concepts of partial program correctness (Definition 6) and always-successful program termination.

\( ^2 \)A final state is typically a state from which an answer can be obtained at low cost.

\( ^3 \)In the program universe thus obtained, \( \text{compute} \) is a total mapping.

\( ^4 \)Due to possible failure to reach a final state in finite number of steps, \( \text{obtain} \) is in general not a total mapping.
Definition 5: Let \( \text{prg} \in \text{PROG} \). A state \( st' \in \text{STATE} \) is said to be reachable from a state \( st \in (\text{STATE} - \text{FIN}) \) using \( \text{prg} \) iff \( \langle st, st' \rangle \in \tau(\text{prg})^n \) for some \( n \geq 0 \), where \( \tau(\text{prg})^0 \) is the identity relation on \( \text{STATE} \) and for each \( m \geq 1 \), \( \tau(\text{prg})^m \) is the composition of \( \tau(\text{prg})^{m-1} \) and \( \tau(\text{prg}) \). Given a state \( st \in (\text{STATE} - \text{FIN}) \), let \( \text{reach}(\text{prg}, st) \) denote the set
\[
\{ st' \mid st' \text{ is reachable from } st \text{ using } \text{prg} \}. \]

Definition 6: A program \( \text{prg} \in \text{PROG} \) is said to be partially correct with respect to a specification \( S \in \text{SPEC} \) iff for any problem \( \text{prb} \in \text{cover}(S) \) such that \( \text{answer}(S, \text{prb}) \) is defined and for any state \( st \in (\text{STATE} - \text{FIN}) \),
\[
\text{makeAns}(st) = \text{answer}(S, \text{prb}).
\]

Next, the notion of always-successful program termination will be introduced.

Definition 7: For any \( \text{prg} \in \text{PROG} \), \( st \in (\text{STATE} - \text{FIN}) \), and \( A \subseteq \text{STATE} \), let the condition
\[
\text{alwaysReach}(\text{prg}, st, A)
\]
be true iff the following conditions are satisfied:

1) There exists no infinite sequence \( [st_0, st_1, st_2, \ldots] \) of states such that
   - \( st_0 = st \),
   - for each \( i \geq 0 \), \( \langle st_i, st_{i+1} \rangle \in \tau(\text{prg}) \),
   - for each \( j \geq 0 \), \( st_j \in (\text{STATE} - A) \).

2) There exists no finite sequence \( [st_0, st_1, st_2, \ldots, st_n] \) of states such that
   - \( st_0 = st \),
   - for each \( i \) such that \( 0 \leq i \leq n-1 \), \( \langle st_i, st_{i+1} \rangle \in \tau(\text{prg}) \),
   - for each \( j \) such that \( 0 \leq j \leq n \), \( st_j \in \text{STATE} - A \),
   - \( st_n \notin \text{dom}(\tau(\text{prg})) \).

Then, for any \( \text{prg} \in \text{PROG} \) and \( S \in \text{SPEC} \), \( \text{prg} \) is said to always successfully terminate with respect to \( S \) iff for any \( \text{prb} \in \text{cover}(S) \),
\[
\text{alwaysReach}(\text{prg}, \text{makeState}(\text{prb}), \text{FIN})
\]
is true.

A necessary and sufficient condition for correctness of a program in a state-transition program universe is now given.

Theorem 1: For any \( \text{prg} \in \text{PROG} \) and \( S \in \text{SPEC} \), \( \text{prg} \) is correct with respect to \( S \) iff the following conditions are satisfied:

1) \( \text{prg} \) is partially correct with respect to \( S \).
2) \( \text{prg} \) always successfully terminates with respect to \( S \).

Proof: Let \( \text{prg} \in \text{PROG} \) and \( S \in \text{SPEC} \).

\((\Rightarrow)\) Assume first that \( \text{prg} \) is correct with respect to \( S \).
Let \( \text{prb} \in \text{cover}(S) \) and assume that \( \text{answer}(S, \text{prb}) \) is defined. Suppose that \( st \) is reachable from \( \text{makeState}(\text{prb}) \) using \( \text{prg} \) and \( st \in \text{FIN} \). Since \( \text{dom}(\tau(\text{prg})) \subseteq (\text{STATE} - \text{FIN}) \), \( st \) does not belong to \( \text{dom}(\tau(\text{prg})) \). As a result, there exists a finite computation \( \text{com} \) of \( \text{prg} \) on \( \text{prb} \) such that \( \text{last}(\text{com}) = st \). Since \( \text{prg} \) is correct with respect to \( S \) and \( \text{answer}(S, \text{prb}) \) is defined, \( \text{obtain}(\text{prg}, \text{com}) = \text{answer}(S, \text{prb}) \). Since \( \text{obtain}(\text{prg}, \text{com}) = \text{makeAns}(\text{last}(\text{com})) \), it follows that \( \text{makeAns}(st) = \text{answer}(S, \text{prb}) \). So \( \text{prg} \) is partially correct with respect to \( S \). Now suppose that \( \text{prg} \) does not always successfully terminate with respect to \( S \). Then there exists \( \text{prb}' \in \text{cover}(S) \) such that \( \text{alwaysReach}(\text{prg}, \text{makeState}(\text{prb}'), \text{FIN}) \) is not true. Consequently, there exists an infinite computation \( \text{com}' \) of \( \text{prg} \) on \( \text{prb}' \) or there exists a finite computation \( \text{com}'' \) of \( \text{prg} \) on \( \text{prb}' \) such that \( \text{last}(\text{com}'') \notin \text{FIN} \). In the first case, \( \text{obtain}(\text{prg}, \text{com}') \) is undefined, and in the second case, \( \text{obtain}(\text{prg}, \text{com}'') \) is undefined. So \( \text{prg} \) is not correct with respect to \( S \), which is a contradiction.

\((\Leftarrow)\) Next assume that \( \text{prg} \) is partially correct with respect to \( S \) and \( \text{prg} \) always successfully terminates with respect to \( S \). Let \( \text{prb} \in \text{cover}(S) \). Obviously, \( \text{compute}(\text{prg}, \text{prb}) \) is defined; moreover, since there always exists at least one computation of \( \text{prg} \) on \( \text{prb} \), \( \text{compute}(\text{prg}, \text{prb}) \neq \emptyset \). Let \( \text{com} \) be a computation of \( \text{prg} \) on \( \text{prb} \). Consider the following two cases:

1) \( \text{com} \) is infinite, and
2) \( \text{com} \) is finite but \( \text{last}(\text{com}) \notin \text{FIN} \).

In both cases, \( \text{alwaysReach}(\text{prg}, \text{makeState}(\text{prb}), \text{FIN}) \) is not true, which is a contradiction. So \( \text{com} \) is necessarily finite and \( \text{last}(\text{com}) \in \text{FIN} \). Hence \( \text{obtain}(\text{prg}, \text{com}) = \text{makeAns}(\text{last}(\text{com})) \). Now assume that \( \text{answer}(S, \text{prb}) \) is defined. Since \( \text{prg} \) is partially correct with respect to \( S \) and \( \text{last}(\text{com}) \) is reachable from \( \text{makeState}(\text{prb}) \) using \( \text{prg} \), \( \text{makeAns}(\text{last}(\text{com})) = \text{answer}(S, \text{prb}) \). Thus \( \text{obtain}(\text{prg}, \text{com}) = \text{answer}(S, \text{prb}) \).

V. Examples of State-Transition Programs

Petri nets [11], term rewriting systems [4], logic programs [9], and constraint logic programs [5], [8] are examples of nondeterministic state-transition programs. In the context of Petri nets [11], a state is a multi-set of places, and each occurrence of a place in a state represents a token in the place. A Petri net is a set of transitions, each of which is a pair of multi-sets of places. When it fires, a transition \( \langle I, O \rangle \) transforms a state \( st \) such that \( I \subseteq st \) into a state \( st' = (st - I) \cup O \). In a term rewriting system [4], a state is a (first-order) term, and a program is a set of rewriting rules. Each rewriting rule determines a binary relation on terms, representing possible state transition it can cause.

In logic programming [9], a state is a set of goals, each of which is a pair \( \langle as, \theta \rangle \), where \( as \) is a sequence of (first-order) atoms and \( \theta \) is a substitution. A logic program specifies a set

\footnote{Such a pair corresponds to a goal \( \neg as \theta \) in Prolog.}
program → (program prog-name (input-vars) body) 
    body → (start-statement statement-list) 
    start-statement → (start : go to label) 
    statement-list → statement-statement-list | statement 
    statement → (label : do variable := expression then go to label) | (label : if condition then go to label else go to label) | (label : halt output-var) 
    expression → integer | variable | succ(variable) | pred(variable) 
    condition → (expression relation expression) 
    relation → = | < | ≤ | ≥ | > | ≠ 
    input-vars → variable-list 
    output-var → variable 
    variable-list → variable variable-list | ε 

Fig. 1. The syntax of NAT programs

Fig. 2. A NAT program

of predicate definitions, and it transforms a state containing a goal \langle as, θ \rangle by unfolding as using one of the predicate definitions, followed by computing the composition of θ and the unifying substitution used in the unfolding operation. In constraint logic programming [5], [8], a state is a set of constraint goals, each of which is a triple \langle as, cs, θ \rangle, where as is a sequence of atoms, cs is a sequence of constraints, and θ is a substitution. A constraint logic program specifies a pair of a set of constraint predicate definitions (each of which is a set of predicate definitions, and it transforms a state containing is a sequence of atoms, \langle m_1, \ldots, m_n \rangle) \in \text{cover}_N(S), \text{answer}_N(S, prb) = f(m_1, \ldots, m_n).

Example 1: Consider a NAT specification \langle add, f \rangle, where f : \mathbb{N} × \mathbb{N} → \mathbb{N} such that for any m_1, m_2 ∈ \mathbb{N}, f(m_1, m_2) = m_1 + m_2. The set cover_N(\langle add, f \rangle) contains infinitely many NAT problems, one of which is \langle add, (3, 2) \rangle. The answer to this problem with respect to \langle add, (3, 2) \rangle is 5. ■

B. The NAT Program Universe

NAT programs are defined by the context-free grammar in Figure 1. An example of a NAT program is given in Figure 2. The NAT program universe \Pi_N is defined as a state transition program universe

\langle PROG_N, PROB_N, STATE_N, IN_N, FIN_N, ANS_N, τ_N, makeState_N, makeAns_N \rangle

as follows:
1) PROG_N is the set of all NAT programs.
2) PROB_N is the set of all NAT programs.
3) A state takes one of the three forms
   a) \langle lab, s_{in} \rangle,
   b) \langle lab, s_{in}, s_{tmp} \rangle, and
   c) \langle lab, result \rangle,
   where lab is a label; s_{in}, s_{tmp} ∈ fseq(\mathbb{N}); and result ∈ \mathbb{N}. STATE_N is the set of all such states. IN_N is the set of all states of the first form. FIN_N is the set of all states of the third form.
4) ANS_N = \mathbb{N}.
5) \tau_N : PROG_N → pow((STATE_N – FIN_N) × STATE_N) is defined as follows. Let prg be a NAT program with input variables \langle u_1, \ldots, u_l \rangle and local variables \langle u_1, \ldots, u_m \rangle. Given a variable w occurring in prg and a state st = \langle lab, s_{in}, s_{tmp} \rangle such that len(s_{in}) = l and len(s_{tmp}) = m, let val prg(w, st) be defined by:
   - If w is u_i for some i ∈ \{1, \ldots, l\}, then val prg(w, st) is the i-th element of s_{in}.
   - If w is u_j for some j ∈ \{1, \ldots, m\}, then val prg(w, st) is the j-th element of s_{tmp}.
   Let Expr be the set of all expressions in NAT programs.
   Let a partial mapping eva prg : Expr × STATE_N → \mathbb{N} be given by: for any state st = \langle lab, s_{in}, s_{tmp} \rangle such that len(s_{in}) = l and len(s_{tmp}) = m,
   - eva prg(n, st) = n for any n ∈ \mathbb{N},

6-NAT” stands for “natural numbers”.

7Local variables are variables that occur in prg but are not input variables.
The answer obtained from this computation is 5. With reference to Theorem 2 on the specification language, if \( \langle s, st' \rangle \) and \( \langle s, st'' \rangle \) belong to \( \tau_N(\text{prg}) \), then \( st' = st'' \). Accordingly, every NAT program is deterministic.

VII. AN EXAMPLE: THE D-RULE COMPUTATION MODEL

Next, formalization of a deterministic state-transition computation model in the domain of first-order terms, called the D-RULE computation model, is demonstrated.

A. Declarative Descriptions

An alphabet \( \Delta = \{K, F, V, R\} \) is assumed, where \( K \) is a set of constants, \( F \) a set of function symbols, \( V \) a set of variables, and \( R \) the union of two disjoint sets of predicate symbols \( R_1 \) and \( R_2 \). Given \( R' \subseteq R \), first-order atoms on \( \langle K, F, V, R' \rangle \) will be referred to as \( \text{atoms} \) on \( R' \). For each \( i \in \{1, 2\} \), let \( A_i \) and \( G_i \) denote the set of all \( i \)-th order atoms on \( R_i \), respectively. Let \( \mathcal{A} = A_1 \cup A_2 \) and \( \mathcal{G} = G_1 \cup G_2 \). Let \( \mathcal{T} \) denote the set of all first-order terms on \( \Delta \) and \( \mathcal{S} \) the set of all substitutions on \( \Delta \). For any substitutions \( \theta, \sigma \in \mathcal{S} \), let \( \theta \circ \sigma \) denote the composition of \( \theta \) and \( \sigma \) (i.e., for any term \( t \in \mathcal{T} \), \( t(\theta \circ \sigma) = (t \theta) \sigma \)). Assume that \( \sigma \) associates to the left.

A definite clause \( cl \) on \( \Delta \) is an expression of the form \( a \leftarrow b_1, \ldots, b_n \), where \( a, b_1, \ldots, b_n \in A \) and \( n \geq 0 \). The atom \( a \) is called the head of \( cl \), denoted by \( \text{head}(cl) \). The set \( \{b_1, \ldots, b_n\} \) is called the body of \( cl \), denoted by \( \text{body}(cl) \).

Given a declarative specification \( D \), a total mapping \( T_D \) on \( \text{pow}(\mathcal{G}) \) is defined by

\[
T_D(G) = \{ \text{head}(cl) \mid (cl \in D) \land (\theta \in \mathcal{S}) \land (\text{body}(cl) \subseteq G) \land (\text{head}(cl) \in \mathcal{G}) \},
\]

for each \( G \subseteq \mathcal{G} \), and the meaning of \( D \), denoted by \( M(D) \), is defined by \( M(D) = \bigcup_{n=1}^{\infty} T_D^n(\emptyset) \), where \( T_D^1(\emptyset) = T_D(\emptyset) \) and \( T_D^n(\emptyset) = T_D(T_D^{n-1}(\emptyset)) \) for each \( n > 1 \).

B. The D-RULE Specification Universe

A D-RULE problem is a singleton set of definite clause from \( R_1 \) to \( R_2 \). A D-RULE specification is a pair \( (D, Q) \), where \( D \) is a declarative description from \( R_1 \) to \( R_1 \) and \( Q \) is a set of D-RULE problems.

The D-RULE specification universe

\[ \Sigma_D = \{ \text{SPEC}_D, \text{PROBD}_D, \text{ANS}_D, \text{COVER}_D, \text{ANSWER}_D \} \]

is defined by:

1. \( \text{SPEC}_D \) is the set of all D-RULE specifications.
2. \( \text{PROBD}_D \) is the set of all D-RULE problems.
3. \( \text{ANS}_D = \text{pow}(\mathcal{G}_2) \).
4. For each \( (D, Q) \in \text{SPEC}_D \), \( \text{COVER}_D((D, Q)) = Q \).

\( D' \) stands for "deterministic" (see Proposition 1).
$D_{rv} = \{ \text{rv}([],[]) \leftarrow, \\
\text{rv}([A],[X,Y]) \leftarrow \text{rv}(X,R), \text{ap}(R,[A],Y), \\
\text{ap}([],[X,X]) \leftarrow, \\
\text{ap}([A],[X,Y],[Z]) \leftarrow \text{ap}(X,Y,Z) \}$

$Q_{fwd} = \{ \{\text{ans}(v) \leftarrow \text{rv}(gl,v)\} | \{\text{gl}\} \text{ is a finite ground list} \} \land \{v \in V\}$

$Q_{bwd} = \{ \{\text{ans}(v) \leftarrow \text{rv}(v,gl)\} | \{\text{gl}\} \text{ is a finite ground list} \} \land \{v \in V\}$

$S_1 = \langle D_{rv}, Q_{fwd} \rangle; S_2 = \langle D_{rv}, Q_{bwd} \rangle; S_3 = \langle D_{rv}, Q_{fwd} \cup Q_{bwd} \rangle$

$prb_1 = \{\text{ans}(X) \leftarrow \text{rv}([1,2],X)\}; prb_2 = \{\text{ans}(X) \leftarrow \text{rv}(X,[1,2])\}$

Fig. 4. Examples of D-RULE specifications and D-RULE problems.

5) For any $\langle D,Q \rangle \in \text{SPEC}_D$ and $prb \in \text{PROB}_D$,

$$answer_D(\langle D,Q \rangle, prb) = T_{prb}(\mathcal{M}(D)).$$

In the sequel, assume that ans (standing for “answer”) is a predicate symbol in $R_2$, and all other predicate symbols appearing in this section belong to $R_1$. The standard Prolog notation for lists will be used.

Example 3: Figure 4 defines a declarative description $D_{rv}$ from $R_1$ to $R_1$ (where $rv$ and $ap$ stand for “for-” and “append”, respectively), D-RULE specifications $S_1$, $S_2$ and $S_3$, and D-RULE problems $prb_1$ and $prb_2$. The problem $prb_1$ is covered by $S_1$ and $S_2$, but not by $S_3$; the problem $prb_2$ is covered by $S_2$ and $S_3$, but not by $S_1$. Referring to $S_1$ (and $S_3$), $prb_1$ represents the query “find every ground instance of $X$ that makes $\text{rv}([1,2],X)$ satisfy the definition of $\text{rv}$ given by $D_{rv}$. The answer to $prb_1$ with respect to $S_1$ (and $S_3$) is the set $T_{prb_1}(\mathcal{M}(D_{rv})) = \{\{\text{ans}([2,1])\}\}$, which means the list $[2,1]$ is the only ground instance of $X$ that satisfies the requirement. The same set is also the answer to $prb_2$ with respect to $S_2$ (and $S_3$).

An answer in $\Sigma_D$ may contain more than one element. For example, let $prb = \{\{\text{ans}(X,Y) \leftarrow \text{ap}(X,Y,[1,2])\}\}$. The answer to $prb$ with respect to $\langle D_{rv}, Q \rangle$, where $prb \in Q$, is the 3-element set $\{\{\text{ans}([],[1,2]),\{\text{ans}([1],[2]),\text{ans}([1,2],[])\}\}$.

C. Built-In Atoms

In addition to usual atoms, built-in atoms are also used in D-RULE programs. Atoms of this kind will now be introduced.

Some terms in $T$ can be evaluated using a predefined procedure. Let a partial mapping $eval_T : T \rightarrow \mathbb{Z}$ be given, where $\mathbb{Z}$ is the set of all integers. Assume that $\text{succ}$ and $\text{pred}$ are function symbols in the alphabet $\Delta$, and for each integer $n$,

- $eval_T(n) = n$
- $eval_T(\text{succ}(n)) = n + 1$
- $eval_T(\text{pred}(n)) = n - 1$

and for any other term $t \in T$, $eval_T(t)$ is undefined.

A built-in atom (for short B-atom) is an expression of one of the forms

- $(v \leftarrow t)$, where $v \in V$ and $t \in T$,
- $(t_1 \text{ rel } t_2)$, where $t_1, t_2 \in T$ and $\text{rel} \in \{=,\neq,<,\leq,\geq\}$

Let $B$ denote the set of all B-atoms. A B-atom will also be evaluated using a predefined procedure, and if the evaluation succeeds, it yields a substitution as a result. It is assumed henceforth that the evaluation result is given by a mapping

$$eval_B : B \rightarrow (S \cup \{\bot, error\})$$

which is defined as follows. For any terms $t, t' \in T$, let $mgu(t, t')$ be the most general unifier of $t$ and $t'$ if they are unifiable, and $mgu(t, t') = \bot$ otherwise. Then, for any $v \in V$ and $t_1, t_2 \in T$,

- if $eval_T(t)$ is defined, then $eval_B(\langle v := t \rangle) = \{\text{eval}(t)\}$,
- otherwise $eval_B(\langle v := t \rangle) = \text{error}$;
- for each rel $\in \{\neq, <, \leq, \geq\}$, if $eval_T(t_1)$ and $eval_T(t_2)$ are defined, then $eval_B(\langle t_1 \text{ rel } t_2 \rangle)$ is defined as
  - $\bot$, if it is true that $eval_T(t_1) \text{ rel } eval_T(t_2)$,
  - $\bot, \text{ otherwise}$;
- and $eval_B(\langle t_1 \text{ rel } t_2 \rangle) = \text{error}$ otherwise.

Based on the mapping $eval_B$, let a mapping

$$eval : fseq(B) \rightarrow (S \cup \{\bot, \text{error}\})$$

be defined by: $eval([]) = \emptyset$, and for any $a \in B$ and $s \in fseq(B)$,

- if $eval_B(a) \in \{\bot, \text{error}\}$, then $eval([a] \cdot s) = eval_B(a)$;
- if $eval_B(a) = \theta \in S$, then $eval([a] \cdot s) = \begin{cases} \theta & \text{if } eval(s) \in S, \\
\bot & \text{if } eval(s) = \emptyset, \\
\text{error} & \text{otherwise.} \end{cases}$

D. D-Rules and D-RULE Programs

A D-rule $r$ is an expression of the form

$$h, \{c_1, \ldots, c_l\} \Rightarrow \{e_1, \ldots, e_m\}, b_1, \ldots, b_n,$$

where $h \in A_1$; $l, m, n \geq 0$; $c_i$ and $e_j$ belong to $B$ for each $i \in \{1, \ldots, l\}$ and $j \in \{1, \ldots, m\}$; and $b_k$ belongs to $A_1$ for each $k \in \{1, \ldots, n\}$. The part $\{c_1, \ldots, c_l\}$ and the part $\{e_1, \ldots, e_m\}$ are called the applicability condition and the execution part, respectively, of $r$; each of them is optional. The D-rule $r$ is said to be applicable to an atom $a \in A_1$ using a substitution $\theta$ iff

- $h \theta = a$, and
- $eval([c_1, \ldots, c_l] \theta) \not\in \{\bot, \text{error}\}$.

A D-RULE program is a finite sequence of D-rules. Figures 5 and 6 illustrate D-RULE programs.

\footnote{The empty set is the identity substitution.}
E. The D-Rule Program Universe

The D-rule program universe $\Pi_D$ is defined as a state transition program universe

$$
\langle \text{PROG}_D, \text{PROB}_D, \text{STATE}_D, \text{IN}_D, \text{FIN}_D, \text{ANS}_D, \tau_D, \text{makeState}_D, \text{makeAns}_D \rangle
$$

as follows:

1) PROG$_D$ is the set of all D-Rule programs.
2) PROB$_D$ is the set of all D-Rule problems.
3) A state is either
   a) an expression of the form $(a \leftarrow bs)$, where $a \in A_2$ and $bs \in f_{seq}(A_1)$, or
   b) $\bot$, which is called the null state.

STATE$_D$ is the set of all such states. A final state is either the null state ($\bot$) or a state of the first form such that $bs$ is the empty sequence. FIN$_D$ is the set of all final states, and IN$_D = \{ \text{STATE}_D - \{ \bot \} \}$. 

4) ANS$_D = \text{pow}(G_2)$.
5) $\tau_D : \text{PROG}_D \rightarrow \text{pow}((\text{STATE}_D - \text{FIN}_D) \times \text{STATE}_D)$ is defined as follows. Given a D-Rule program $\text{prg}$ and a state

   $st = (a \leftarrow [b] \cdot bs)$,

   $\langle st, st' \rangle \in \tau_D(\text{prg})$ iff there exist a D-Rule

   $r = (h, \{c_1, \ldots, c_l\} \Rightarrow \{e_1, \ldots, e_m\}, b_1, \ldots, b_n)$

in $\text{prg}$ and a renaming substitution $\gamma$ for $r$ such that

- no variable occurring in $r\gamma$ occurs in $st$,
- $r\gamma$ is applicable to $b$ using $\theta \in S$, and
- for any D-Rule $r'$ that occurs before $r$ in $\text{prg}$, there exists no renaming substitution $\gamma'$ for $r'$ such that

   $r'\gamma'$ is applicable to $b$,

and $st'$ is obtained as follows:

   a) let $\sigma = \text{eval}(\{c_1, \ldots, c_l\}(\gamma \circ \theta))$,
   b) if $\text{eval}(\{e_1, \ldots, e_m\}(\gamma \circ \theta \circ \sigma) = \rho \in S$, then $st'$ is the state

      $$(a(\sigma \circ \rho) \leftarrow [b_1, \ldots, b_n](\gamma \circ \theta \circ \sigma \circ \rho) \cdot bs(\sigma \circ \rho)),$$
   c) if $\text{eval}(\{e_1, \ldots, e_m\}(\gamma \circ \theta \circ \sigma)) = \bot$, then $st' = \bot$.

6) For any problem $\text{prb} = \{(a \leftarrow [b_1, \ldots, b_n])\}$,

   $\text{makeState}_D(\text{prb}) = (a \leftarrow [b_1, \ldots, b_n])$.

7) For any final state $st$ of the form $(a \leftarrow [])$,

   $\text{makeAns}_D(st) = \{a\theta \mid (\theta \in S) \& (a\theta \in G)\}$,

   and $\text{makeAns}_D(\bot) = \emptyset$.

Example 4: Consider the D-Rule problem $\text{prb}_2$ in Figure 4. A computation of the D-Rule program in Figure 5 on $\text{prb}_2$ is shown in Figure 7. The answer obtained from this computation is the list $[2, 1]$. Referring to Example 3, this obtained answer coincides with the answer to $\text{prb}_2$ with respect to $S_2$. 

Example 5: Figure 8 shows a computation of the D-Rule program in Figure 6 on the D-Rule problem $\{(\text{ans}(X) \leftarrow \text{add}(3, 2, X))\}$. The obtained answer is $5$. 

Let an equivalence relation $\sim_D$ on $\text{COM}_D$ be defined by:

for any $\text{com}$, $\text{com}' \in \text{COM}_D$, $\text{com} \sim_D \text{com}'$ iff $\text{com}'$ is a variant of $\text{com}$, i.e., there exists a renaming substitution $\gamma$ for $\text{com}'$ such that $\text{com} = \text{com}'\gamma$. It follows from the definition of $\tau_D$ that:

**Proposition 1:** Every D-Rule program is deterministic up to $\sim_D$; i.e., for any D-Rule program $\text{prg}$ and D-Rule problem $\text{prb}$, if $\text{com}$ and $\text{com}'$ are computations of $\text{prg}$ on $\text{prb}$, then $\text{com} \sim_D \text{com}'$. 

VIII. Embedding the NAT Computation Model in the D-Rule Computation Model

To demonstrate a comparison between computation models based on the presented framework, the concept of an embedding mapping is introduced, using the NAT computation model and the D-Rule computation model as examples.

Basically, an embedding mapping from the NAT computation model to the D-Rule computation model is a 4-tuple $(\alpha_{\text{prb}}, \alpha_{\text{spc}}, \alpha_{\text{prog}}, \alpha_{\text{state}})$, where

- $\alpha_{\text{prb}} : \text{PROB}_N \rightarrow \text{PROB}_D$,
- $\alpha_{\text{spc}} : \text{SPEC}_N \rightarrow \text{SPEC}_D$,
- $\alpha_{\text{prog}} : \text{PROG}_N \rightarrow \text{PROG}_D$,
- $\alpha_{\text{state}} : \text{STATE}_N \rightarrow \text{STATE}_D$.

that preserves the structure of $(\Sigma_N, \Pi_N)$. It will be shown that such a mapping always preserves the correctness of
programs in $\Pi_N$ (Theorem 2). As a result, the notion of an embedding mapping provides a means of rigorously comparing the correct-program spaces of computation models. In general, if there exists an embedding mapping from one computation model to another computation model but not vice versa, then the correct-program space of the latter model is larger than that of the former one; and, accordingly, the latter model provides a better structure for searching for correct and efficient programs, and for discussion of program synthesis.

An embedding mapping $\langle \alpha_{prb}, \alpha_{spec}, \alpha_{prog}, \alpha_{st} \rangle$ from the NAT computation model to the D-RULE computation model is defined by Subsections VIII-A–VIII-C. Its basic structure-preserving properties are provided in Subsection VIII-D along with some related results. The proof of the resulting correctness-preserving property is given in Subsection VIII-E.

A. Mapping Problems and Specifications: $\alpha_{prb}$ and $\alpha_{spec}$

Referring to the sets NAME and $R_1$ of Sections VI and VII, respectively, assume that NAME $\subseteq R_1$. Let $\alpha_{prb} : \text{PROB}_N \rightarrow \text{PROB}_D$ and $\alpha_{spec} : \text{SPEC}_N \rightarrow \text{SPEC}_D$ be defined as follows:

1) For any NAT problem $prb = \langle \text{name}, \langle m_1, \ldots, m_n \rangle \rangle$, 
$$\alpha_{prb}(prb) = \{ \langle \text{ans}(X) \leftarrow \text{name}(m_1, \ldots, m_n) \rangle \}.$$ 

2) For any NAT specification $S = \langle \text{name}, f \rangle$, where $f : \mathbb{N}^n \rightarrow \mathbb{N}$ and $n \geq 1$, $\alpha_{spec}(S) = \{ D, Q \}$, where
- $D$ is the set of all unit clauses
  $$\text{name}(m_1, \ldots, m_n, \text{result}) \leftarrow$$
  such that $\langle m_1, \ldots, m_n \rangle \in \text{dom}(f)$ and $\text{result} = f(m_1, \ldots, m_n)$,
- $Q = \{ \alpha_{prb}(prb) : prb \in \text{cover}_N(S) \}$.

Example 6: Let $prb$ be the NAT problem $\langle \text{add}, \langle 3, 2 \rangle \rangle$. Then $\alpha_{prb}(prb) = \{ \langle \text{ans}(X) \leftarrow \text{add}(3, 2, X) \rangle \}$. Considering the NAT specification $S_{\text{add}}$ of Example 1, $\alpha_{spec}(S_{\text{add}}) = \{ D_{\text{add}}, Q_{\text{add}} \}$, where $D_{\text{add}}$ is the set consisting of all unit clauses $\langle \text{add}(m_1, m_2, n) \leftarrow \rangle$ such that $m_1, m_2 \in \mathbb{N}$ and $n = m_1 + m_2$, and $Q_{\text{add}}$ is the set consisting of all D-RULE problems $\{ \langle \text{ans}(X) \leftarrow \text{add}(m_1', m_2', X) \rangle \}$ such that $m_1', m_2' \in \mathbb{N}$. 

B. Mapping Programs: $\alpha_{prog}$

Next, $\alpha_{prog} : \text{PROG}_N \rightarrow \text{PROG}_D$ is defined. Let $prg$ be a NAT program with the program name $\text{progNm}$ and with input variables $v_1, \ldots, v_l$ and local variables $u_1, \ldots, u_m$, in order of appearance. Without loss of generality, assume that there is no occurrence of the variable $X$ in any $prg$. Then $\alpha_{prog}(prg)$ is obtained by successively mapping statements in $prg$ into D-rules as follows.

1) A statement of the form $(\text{start} : \text{do go to lab})$ is mapped into a D-rule
   $$\text{progNm}(v_1, \ldots, v_l, X) \Rightarrow \text{lab}(v_1, \ldots, v_l, \text{zero}_l, \ldots, \text{zero}_m, X),$$
   where each of $\text{zero}_1, \ldots, \text{zero}_m$ is the integer zero.

2) A statement of the form $(\text{lab} : \text{do w := expr then go to lab'})$ is mapped into a D-rule
   $$\text{lab}(v_1, \ldots, v_l, u_1, \ldots, u_m, X) \Rightarrow \{ \langle \text{w := expr} \rangle, \text{lab'}(v_1, \ldots, v_l, u_1, \ldots, u_m, X) \theta \},$$
   where $\theta$ is a variable that is distinct from $w, v_1, \ldots, v_l, u_1, \ldots, u_m$ and $X$, and $\theta$ is the substitution $\{ w/w' \}$.

3) A statement of the form
   $$(\text{lab} : \text{if rel expr} \text{ then go to lab'} \text{ else go to lab''}),$$
   where rel $\in \{=, <, \leq \}$, is mapped into two successive D-rules
   $$\text{lab}(v_1, \ldots, v_l, u_1, \ldots, u_m, X), \{ \langle \text{expr rel expr} \rangle \} \Rightarrow \text{lab'}(v_1, \ldots, v_l, u_1, \ldots, u_m, X),$$
   $$\text{lab}(v_1, \ldots, v_l, u_1, \ldots, u_m, X), \{ \langle \text{expr rel expr} \rangle \} \Rightarrow \text{lab''}(v_1, \ldots, v_l, u_1, \ldots, u_m, X),$$
   where $=, <$ and $\leq$ are $\neq, \neq$ and $\leq$, respectively.

Example 7: From the NAT program in Figure 2, the D-RULE program in Figure 6 is obtained using $\alpha_{prog}$. 

C. Mapping States: $\alpha_{st}$

Finally, $\alpha_{st} : \text{STATE}_N \rightarrow \text{STATE}_D$ is defined by: for any state $st$ in $\text{STATE}_N$,
- if $st = \{ \langle \text{lab}, \langle m_1, \ldots, m_i \rangle \rangle \}$, then
  $$\alpha_{st}(st) = \{ \langle \text{ans}(X) \leftarrow \langle \text{lab}(m_1, \ldots, m_i, X) \rangle \rangle \};$$
- if $st = \{ \langle \text{lab}, \langle m_1, \ldots, m_i \rangle, \langle m'_1, \ldots, m'_n \rangle \rangle \}$, then
  $$\alpha_{st}(st) = \{ \langle \text{ans}(X) \leftarrow \langle \text{lab}(m_1, \ldots, m_i, m'_1, \ldots, m'_n, X) \rangle \rangle \};$$
- if $st = \{ \langle \text{lab}, m \rangle \}$, then $\alpha_{st}(st) = \{ \langle \text{ans}(m) \leftarrow [\rangle \}.$

Example 8: By applying $\alpha_{st}$ to each state occurring in the computation in Figure 3, the computation in Figure 8 is obtained.

D. Structure-Preserving Properties and Related Results

Structure-preserving properties of $\langle \alpha_{prb}, \alpha_{spec}, \alpha_{prog}, \alpha_{st} \rangle$ and some related results will now be given. Propositions 2–6 follow directly from the definitions of $\alpha_{prb}$, $\alpha_{spec}$, $\alpha_{prog}$, and $\alpha_{st}$ in Subsections VIII-A–VIII-C.

Proposition 2: For any $S \in \text{SPEC}_N$,
$$\alpha_{prb}(\text{cover}_N(S)) = \text{cover}_D(\alpha_{spec}(S)).$$

Proposition 3: For any $S \in \text{SPEC}_N$, and $prb \in \text{cover}_N(S)$,
$$\text{answer}_N(S, prb) = m \Rightarrow \text{answer}_D(\alpha_{spec}(S), \alpha_{prb}(prb)) = \{ m \}. $$
Proposition 4: For any $prb \in \text{PROB}_N$,
$$\alpha_{st}(\text{makeState}_N(prb)) = \text{makeState}_D(\alpha_{prb}(prb)).$$

Proposition 5: For any $st \in \text{FIN}_N$,
$$\text{makeAns}_N(st) = m \iff \text{makeAns}_D(\alpha_{st}(st)) = \{m\}.$$  

Proposition 6: For any $prg \in \text{PROG}_N$, $st \in (\text{STATE}_N - \text{FIN}_N)$ and $st' \in \text{STATE}_N$,
$$\langle st, st' \rangle \in \tau_N(prg) \iff \langle \alpha_{st}(st), \alpha_{st}(st') \rangle \in \tau_D(\alpha_{prg}(prg)).$$

Proposition 6 yields the next two results.

Proposition 7: For any $prg \in \text{PROG}_N$ and $st \in \text{STATE}_N$,
$$\alpha_{st}(\text{reach}(prg, st) \cap \text{FIN}_N) = \text{reach}(\alpha_{prg}(prg), \alpha_{st}(st)) \cap \text{FIN}_D.$$  

Proposition 8: For any $S \in \text{SPEC}_N$ and $prg \in \text{PROG}_N$, $prg$ always successfully terminates with respect to $S$ if $\alpha_{prg}(prg)$ is partially correct with respect to $\alpha_{spec}(S).$

E. Correctness-Preserving Property

Preservation of partial correctness will be shown first.

Proposition 9: (Preservation of partial program correctness) For any $S \in \text{SPEC}_N$ and $prg \in \text{PROG}_N$, $prg$ is partially correct with respect to $S$ if $\alpha_{prg}(prg)$ is partially correct with respect to $\alpha_{spec}(S).$

Proof: Let $S \in \text{SPEC}_N$ and $prg \in \text{PROG}_N$. Note first that $\text{answer}_N(S, prb)$ is defined for each $prb \in \text{cover}_N(S)$, and $\text{answer}_D(\alpha_{spec}(S), prb')$ is defined for each $prb' \in \text{cover}_D(\alpha_{spec}(S))$. It follows that:

$prg$ is partially correct with respect to $S$
$$\iff \forall prb \in \text{cover}_N(S),$$
$$\forall st \in (\text{reach}(prg, \text{makeState}_N(prb)) \cap \text{FIN}_N)$ : 
$$\text{makeAns}_N(st) = \text{answer}_N(S, prb)$$
$$\iff \forall prb \in \text{cover}_N(S),$$
$$\forall st' \in \alpha_{st}(\text{reach}(prg, \text{makeState}_N(prb)) \cap \text{FIN}_N)$ : 
$$\text{makeAns}_D(st') = \text{answer}_D(\alpha_{spec}(S), \alpha_{prb}(prb))$$
(by Propositions 3 and 5)
$$\iff \forall prb \in \text{cover}_N(S),$$
$$\forall st' \in (\text{reach}(\alpha_{prb}(prg), \alpha_{st}(\text{makeState}_N(prb))) \cap \text{FIN}_D)$ : 
$$\text{makeAns}_D(st') = \text{answer}_D(\alpha_{spec}(S), \alpha_{prb}(prb))$$
(by Proposition 7)
$$\iff \forall prb' \in \text{cover}_2(\alpha_{spec}(S)),$$
$$\forall st' \in (\text{reach}(\alpha_{prb}(prg), \text{makeState}_D(prb')) \cap \text{FIN}_D)$ : 
$$\text{makeAns}_D(st') = \text{answer}_D(\alpha_{spec}(S), prb')$$
(by Proposition 4)
$$\iff \alpha_{prb}(prg)$ is partially correct with respect to $\alpha_{spec}(S).$  

Preservation of program correctness (Theorem 2) follows immediately from the necessary and sufficient condition for program correctness established in Section IV (Theorem 1), preservation of successful termination (Proposition 8), and preservation of partial correctness (Proposition 9).

Theorem 2: (Preservation of program correctness) For any $S \in \text{SPEC}_N$ and $prg \in \text{PROG}_N$, $prg$ is correct with respect to $S$ if $\alpha_{prg}(prg)$ is correct with respect to $\alpha_{spec}(S).$

IX. CONCLUDING REMARKS

By giving mathematical characterization of state-transition computation models, this paper formalizes the common structure of most, if not all, computation models ever considered in the literature, including sequential and parallel computation models. A necessary and sufficient condition for program correctness on this class of computation models is established. Formalization of two specific state-transition computation models using the presented framework is illustrated, i.e., the NAT computation model and the D-Rule computation model. An embedding mapping from the NAT computation model to the D-Rule computation model is illustrated. The existence of this mapping implies that the D-Rule computation model provides a better structure for construction of and searching for programs that are correct with respect to a given specification. The presented framework provides an important step towards development of a general theory for analyzing correctness-related properties of programs and for comparing computation models from the viewpoint of program synthesis. Based on this framework, research on rigorous comparisons between the logic programming model [9], the constraint logic programming model [5], [8], and the equivalent transformation computation model [1] is well underway.

REFERENCES