On Describing Terminating Algebraic Specifications Based on Their Models

Masaki Nakamura, Kazuhiro Ogata and Kokichi Futatsugi

Abstract—OBJ algebraic specification languages support automated equational reasoning based on term rewriting systems (TRSs) for specification verification. Termination is one of the most important properties of TRSs. Terminating TRSs guarantee that any equational reasoning terminates in finite times. Although termination is an undecidable property, several sufficient conditions have been proposed, and several termination provers have been developed. In this study, we focus on a way to describe terminating algebraic specifications, that is, the corresponding TRSs are terminating. Existing termination provers may inform us whether a given specification is terminating or not. However, they do not give a guideline to describe terminating specifications. We propose a model-based method for describing terminating specifications. In our method, a model of a given specification can be used for proving its termination. Since specifiers are describing a specification while thinking its model in their mind, our model-based termination methods are suitable for algebraic specifications.

Index Terms—Algebraic specification, Term rewriting, OBJ languages, Termination.

I. INTRODUCTION

OBJ languages [5], [6], [3], [10] are algebraic specification languages, based on order-sorted equational logic. OBJ languages support many advanced features, e.g., module system, typing system with ordered sorts, mix-fix syntax, for describing specifications, and a powerful interactive theorem proving system based on term rewriting systems (TRSs). Executability is an important feature of OBJ languages. Since most OBJ specifications are assumed to be executed for verifying desired properties, the knowledge about TRSs may help us to describe executable specifications. Termination is one of the most important properties of TRSs, which guarantees that execution of the specification must terminate in finite times. Properties of TRSs are important not only verifying specification but also describing specifications correctly. For example, termination is strongly related to the existence of a solution. Confluence is another important property of TRSs, which is related to the uniqueness of a solution. Reducibility of TRSs helps us to check whether a function is well-defined for all elements of its domain

\[ f \in \Sigma \]

Termination is considered most important among those fundamental properties. Although confluence and reducibility are also undecidable properties in general, they are decidable and automatically provable when a given TRS is terminating. In this study, we focus on describing algebraic specifications whose TRSs are terminating.

In algebraic specifications, we define functions by a set of equations (axioms). For example, consider a function \( \text{even} \) which take a natural number and returns \( \text{true} \) if it is even, otherwise \( \text{false} \). The function \( \text{even} \) (and \( \text{odd} \)) can be defined by the following set of equations:

\[
\begin{align*}
\text{even}(0) &= \text{true}, \\
\text{odd}(0) &= \text{false}, \\
\text{even}(n + 1) &= \text{odd}(n), \\
\text{odd}(n + 1) &= \text{even}(n).
\end{align*}
\]

We can compute the value of \( \text{even}(n) \) by applying the equations in the left-to-right manner, like \( \text{even}(3) = \text{odd}(2) = \text{even}(1) = \text{odd}(0) = \text{false} \). Computation of \( \text{even}(n) \) for any \( n \) terminates since the argument of \( \text{even} \) or \( \text{odd} \) is strictly decreasing by application of an equation, and will eventually reach zero. Consider another example: the reverse function on lists. The equations \( \text{rev}(\text{nil}) = \text{nil} \) and \( \text{rev}([e]; \text{nil}) = \text{rev}([\text{nil}]; [e]) \) define the reverse function \( \text{rev} \) on lists, where \( \text{nil} \) is the empty list, \( [e] \) is the list whose head element is \( e \) and tail list is \( \text{nil} \), and @ is the list concatenation. The reverse of \( [1; 2; 3; \text{nil}] \) can be computed as follows:

\[
\begin{align*}
\text{rev}([1; 2; 3; \text{nil}]) &= \text{rev}([2; 3; \text{nil}]; \text{nil}) \\
&= \text{rev}([3; \text{nil}]; [2; \text{nil}]; \text{nil}) \\
&= \text{rev}([\text{nil}; [3; \text{nil}]; [2; \text{nil}]; \text{nil}]) \\
&= \text{nil} \\
&= 3; 2; 1; \text{nil}.
\end{align*}
\]

The computation terminates since the length of the argument list of \( \text{rev} \) is strictly decreasing. Intuitively, the above examples are terminating since application of each equation may decrease something strictly. OBJ specifications provide a way to describe abstract data type. We can describe ordinary basic data types like numbers, strings, etc., and more complicated types like lists, sets, collection types and so on. When describing large and complex systems consisting of several data types and functions, it is not easy to describe equations where any combination of their application must terminate. In this study, we propose a model-based method for proving termination, which gives us a mathematical evidence that any computation must terminate.

II. PRELIMINARIES

In this section, we introduce the notion of algebraic specifications [3] and term rewriting systems (TRSs) [13]. Though we give CafeOBJ specifications for explanation, our approach does not restrict the target to CafeOBJ, and can also be applied to other OBJ languages.

A. Algebraic specifications

An algebraic specification consists of signature and axioms. A signature \( \langle S, \Sigma \rangle \) (abbr. \( \Sigma \)) consists of a set \( S \) of sorts, a quasi-order \( \leq \) on \( S \), and a set \( \Sigma \) of operation symbols defined on the sorts\(^1\). An operation symbol \( f \in \Sigma \) is a binary relation that is transitive and reflexive.

\(^1\)A binary relation is a quasi-order if it is transitive and reflexive.
has its rank. A rank consists of an arity and a sort. An arity is a sequence of sorts. An OBJ specification consists of modules. The following is a CafeOBJ module NAT+:

\[
\begin{align*}
\text{mod} & \mathtt{NAT}\{ \\
& [\mathtt{Zero \, NzNat < Nat}] \\
& \text{op } 0 \rightarrow \mathtt{Zero} \\
& \text{op } s\_ \rightarrow \mathtt{NzNat} \\
& \text{op } \_+\_ \rightarrow \mathtt{NzNat} \rightarrow \mathtt{Nat} \\
& \text{vars } M \ : \ \mathtt{Nat} \\
& \text{eq } N + 0 = N . \\
& \text{eq } M + s \, N = s(M + N) .
\end{align*}
\]

\(\mathtt{NAT}\) consists of a single module. Roughly speaking, \(\mathtt{NAT}\) denotes natural numbers with the addition function. The symbol 0 stands for zero, and \(s\_\) stands for the Peano style successor function. The terms 0, s 0, s s 0, ... are regarded as 0, 1, 2, ... respectively. In \(\mathtt{NAT}\), the sorts \(\mathtt{Zero, NzNat, Nat}\) and \(\mathtt{Nat}\) are declared with the subset relation \(\mathtt{Zero < NzNat < Nat}\). In \(\mathtt{NAT}\), the set \(\mathtt{Nat}\) of sorts is \{\(\mathtt{Zero, NzNat, Nat}\}\} and the partial order \(\leq\) is the reflexive and transitive closure of \(<\). That is, \(\leq_{\mathtt{NAT}} = \{\{\mathtt{Zero, Zero}\}, \{\mathtt{NzNat, NzNat}\}, \{\mathtt{Nat, Nat}\}, \{\mathtt{Zero, Nat}\}, \{\mathtt{NzNat, Nat}\}\}\}.

In CafeOBJ, an operation symbol \(f\) is declared with its rank as \(\text{op } f : \text{arity} \rightarrow \text{sort}\). An operation symbol is called a constant when the arity is empty. Module \(\mathtt{NAT}\) has the declarations of operation symbols 0, \(s\_\) and \(\_+\_,\_\) where 0 is a constant.

A \((\Sigma, X)\)-term is a tree node whose operation symbols are in \(\Sigma\) and leaves are variables in \(X\). We may omit the prefix \((\Sigma, X)-\). For a given signature \((\Sigma, \leq, \Sigma)\) and an \(S\)-sorted set \(X\) of variables, the \(S\)-sorted set \(T_{\Sigma}(X)\) (abbr. \(T_{\Sigma}\)) of \((\Sigma, X)\)-terms is defined as the smallest set satisfying that (1) \(X \subseteq T_{\Sigma}\) for each \(x \in S\), (2) \(T_{\Sigma} \subseteq T_{\Sigma}\) for each \(s \leq s'\), and (3) \(f(t_1, \ldots , t_n) \in T_{\Sigma}\) for each \(f \in \Sigma\) with the arity \(s_1s_2\cdots s_n\) and the sort and, terms \(t_i \in T_{\Sigma}\) (\(i \in [\bar{n}]\)).

A term \(t\) belonging to \(T_{\Sigma}\) is called a term of the sort \(s\). In CafeOBJ, we can indicate the arguments positions of equations in the axiom part. For a signature \(\Sigma = (\leq, \leq, \Sigma)\), \(\Sigma\)-algebra \(M\) consists of an \(S\)-sorted carrier set \(M\) where \(M_s \subseteq M_{s'}\) for each \(s < s'\), and functions \(M_f : M_{s_1} \times \cdots \times M_{s_n} \rightarrow M_r\) for each \(op\, f : s_1 \cdots s_n \rightarrow r\) in \(\Sigma\). For a \(\Sigma\)-algebra \(M\) and an assignment \(\alpha : X \rightarrow M\), a map \(\bar{\alpha} : T \rightarrow M\) is defined as follows: \(\bar{\alpha}(x) = \alpha(x)\) for each \(x \in X\) and \(\bar{\alpha}(f(t_1, \ldots , t_n)) = M_f(\bar{\alpha}(t_1), \ldots , \bar{\alpha}(t_n))\) for each \(f(t_1, \ldots , t_n) \in T\). We may use \(\alpha\) instead of \(\bar{\alpha}\) if no confusion arises. A \(\Sigma\)-algebra \(M\) satisfies an equation \(l = r\) if \(\bar{\alpha}(l) = \alpha(r)\) for each assignment \(\alpha\), denoted by \(M_l = M_r\). For a given set \(E\) of equations, a \((\Sigma, E)\)-algebra is a \(\Sigma\)-algebra which satisfies all equations in \(E\).

An initial model \(M\) of \(SP\) is a kind of the smallest \((\Sigma, E)\)-algebras, which means that each element \(e \in M\) corresponds to some ground (variable-free) term \(t \in T_{\Sigma}(\emptyset)\) (no junk) and \(Mt = M_e\) implies \(t = E t'\) for all \(t, t' \in T_{\Sigma}(\emptyset)\) (no confusion). The set of all models of \(SP\) is denoted by \(M(\Sigma)\) and the set of all initial models of \(SP\) is denoted by \(IM(\Sigma)\). There are two kinds of denotations of CafeOBJ modules: the tight denotation (mod!) and the loose denotation (mod*). The denotation of a basic CafeOBJ module \(SP\), denoted by \([SP]\), is defined as follows: When mod! \(SP\) the \(\nu SP\) \([SP]\) = \(IM(\Sigma)\) and when mod* \(SP\) \([SP]\) = \(M(\Sigma)\), where a basic module is a module without any import. By using module imports, we can describe large and complex systems easily. For example, when a module \(SP\) imports \(SP\) with the protect mode, each denotational model \(M' \in [SP']\) includes a model \(M' = [SP]\) of the imported module as it is, denoted by mod* \(SP' [\nu SP]\) ...

For example, we can describe specifications of data types, and a system specification imports them with the protected mode. We omit details of denotation of modules including module imports (See the reference [3]).

Consider the following \(\Sigma_{\mathtt{NAT}}\)-algebras \(B, N, Z, N^+\).

- \(B\) is a Boolean algebra: \(B_{\text{true}} = \{\text{false}\}\), \(B_{\text{false}} = \{\text{true}\}\), \(B_{\text{true}} = \{\text{false}\}\), \(B_{\text{false}} = \{\text{true}\}\).
- \(N\) is an algebra of natural numbers: \(N_{\text{zero}} = \{0\}\), \(N_{\text{nat}} = \{1, 2, \ldots\}\), \(N_{\text{nat}} = \{0, 1, 2, \ldots\}\), \(N^0 = 0\), \(N^+ = \{x \in \mathbb{R}^+ \mid x + 1\}\), \(N^+ = \{x \in \mathbb{R}^+ \mid x + 1\}\).
- \(Z\) is an algebra of integers: \(Z\) is same with \(N\) except \(Z_{\text{nat}} = \{\cdot \ldots, -1, 0, 1, 2, \ldots\}\).
- \(N^+\) is another algebra of natural numbers: \(N^+_e\) is same with \(N^\_e\) except \(N^+_e = \{x \in \mathbb{R}^+ \mid x\}^{\times}\).

All \(B, N, Z, N^+\) are \(\Sigma_{\mathtt{NAT}}\)-algebras. Only \(N^+\) is not a \((\Sigma_{\mathtt{NAT}}\), mod\+)\)-algebra since the equations are not satisfied, for example, \(N_{\text{nat}}^{n+1} = a(0) \times 0 = 0 \neq a(0) = B_0\) for some \(a\) which assigns \(N\) to a positive integer. A specification \(\mathtt{NAT}\) denotes \(N\), i.e. \(N \in [\mathtt{NAT}]\). \(Z\) has negative integers (junk) and \(B\) interpret all positive integers to one element true (confusion), that is, \(B_{\text{true}} = B_{\text{false}} = \text{true}\) though \(s \neq 0 \neq E \neq 0\). Thus, \(Z, B \notin [\mathtt{NAT}]\).

C. Term rewriting systems

An OBJ specification is executable, that is, some term may be reduced into its normal form automatically. The execution is based on a term rewriting system (TRS), which is a set of

\[t \rightarrow^* t'\] means that the equation can be reduced from \(E\).

In this paper, the contents of \(SP\) may be written like \(X_{SP}\). For example, \(X_{SP}\) and \(E_{SP}\) are the signature and the set of all equations of a specification (or a module) \(SP\) respectively.
rewrite rules. In a TRS, an equation is regarded as a left-to-right rewrite rule. A term is rewritten by replacing an instance of the left-hand side of an equation with the corresponding instance of the right-hand side. A term is reduced by applying rewrite rules to a given term until it cannot. The following is an example of reducing the term s 0 + s 0 0 in NAT+:

CafeOBJ> red in NAT+ : s 0 + s 0 .
M -- reduce in NAT+ : s 0 + s (s 0)
M (s (s 0)) : NzNat.

The following is the trace of the above reduction showed by CafeOBJ system:

s 0 + s 0 0 → s (s 0 + s 0) → s s (s 0 + 0) → s s s 0. The underlined subterms are matched with left-hand sides of equations, and replaced with the corresponding right-hand sides. The first and second rewrites are obtained by the second equation eq M + s N = s (M + N) and the last rewrite is obtained by the first equation eq M + s N = s (M + N). The input and output terms of the CafeOBJ reduction command are equivalent in all models [3]. Thus, the above execution is a proof of 1 + 2 = 3 in NAT+. The rewrite relation obtained by an equation e is denoted by →e, and the rewrite relation obtained by E is denoted by →E. The reflexive and transitive closure of → is denoted by →*. The above reduction can be written as s 0 + s 0 0 →*M s 0 s 0 0.

D. Termination

A specification SP = (Σ, E) (or a TRS E) is terminating if there is no infinite rewrite sequence [13]. Let Eθ be the set obtained by adding eq s (0 + N) = 0 + s N to NAT+. Then, Eθ is not terminating since there is an infinite rewrite sequence s (0 + 0) → Eθ. 0 + s 0 s 0 → Eθ. s (0 + 0) → Eθ. . . . Several useful proof methods for termination have been proposed and several termination provers have been developed. We can use those tools to prove a given specification to be terminating. We introduce two classical approaches and one recent powerful notion for termination.

1) Recursive Path ordering: Recursive path ordering (RPO) is one of the most classical syntactic termination methods. Let Σ be a signature, A a set of variables, and ≼p⊂Σ × Σ a partial order on symbol terms. The RPO ≼p⊂ Σ, X ⊂ T (Σ, X) is defined as follows:

- t ≼p⊂ t′ \iff t = f(t_1) and
- 1) \exists i \in \{0, 1\} t_i \rhd \sim_{p_1} t_1, or
- 2) t′ = g(P_i), f \triangleright g and \forall j \in \{0, 1\} t \rhd \sim_{p_1} t_j, or
- 3) t′ = g(P_i), f \triangleright g and \{f(t_1) \rhd \sim_{p_m} t_1\}.

where a \triangleright b \iff a > b \land a \neq b, a \triangleright b \iff a > b \land a \triangleright b. [\ldots \}] stands for a multiset. The partial order \sim_{p_m} is a multiset order w.r.t. >p. The following property holds.

Proposition 2.1: [13] Let E be a set of equations. If there exists \Sigma such that t \rhd \sim_{p_1} r for each l \lneq r \in E, then E is terminating.

2) Polynomial interpretations: In semantic methods, termination is proved based on a well-founded monotone compatible Σ-algebra, that is, a (\Sigma, \leq, \Sigma)-algebra together with a partial order on M2 which is well-founded, monotone and compatible with E. A partial order \triangleright is well-founded if there is no infinite decreasing chain a1 \triangleright a2 \triangleright . . . . It is monotone if \mid F(t(a_1)) \triangleright F(t(b_1)) \rangle whenever f_1 \in \Sigma, \exists i \in \{0, 1\} a_i \triangleright b_i, and \forall j \neq i, a_j = b_j. It is compatible with E if E(M_1, M_2) whenever l = r \in E.

Proposition 2.2: [13] A specification E is terminating if and only if there exists a well-founded monotone compatible Σ-algebra.

We show that termination of NAT+ by giving a Σ-algebra M where operation symbols are interpreted into polynomials on natural numbers. Let M_0 = 1, M_1(x) = x + 1 and M_2(x, y) = 2y + 1. Let a be an assignment such that a(n) = n and a(M) = m. Then the first equation in NAT+ is ordered by \triangleright: M_0 \triangleright M_0 + (n, M_0) = n + 3 > n. So is the second equation: M_1 \triangleright M_1 + (m, M_1) = m + 2n + 3 > m + 2n + 2 = M_2(M_1(n, n)) = M_0(n, n). Thus, M is a proof of termination of NAT+.

3) Dependency pairs: Recently a landmark in termination methods has been proposed, called the dependency pair (DP) method [1]. A dependency pair is a pair of terms obtained from an equation, which is essential for analyzing termination. Combinations of the DP method and classical methods like RPO, Polynomial ordering, etc. extend a class of TRS which can automatically be proved terminating. Most of the active termination tools adopt the DP method.

A root symbol, i.e. the symbol at the root position, of a left-hand side of an equation is called a defined symbol. For example, \# + \# is a defined symbol and 0 and s_0 are not. A rename of an operation symbol f is denoted by f^#.

We assume that f^# is not included in the original signature Σ. The term whose root symbol is renamed is denoted by t^#.

For example, t^# = s^#(0, N) when t = 0 \triangleright N. For the readability, we write f^#(\ldots) even if f is declared with the underlines, like f_{\underline{\ldots}}. Let l = r \in E be an equation. A dependency pair is a pair (ln^#, ur^#) of terms where u is a subterm of r whose root symbol is a defined symbol [1]. The all dependency pairs obtained from E is denoted by DP(E). A chain of dependency pairs is a (possibly infinite) sequence (ln^#, ur^#) \{(i = 0, 1, 2, \ldots)\} of dependency pairs in DP(E) whose variables are distinct such that dependency pairs do not share variables and there is a substitution θ such that ur^#θ \rhd \rhd E ln^#θ for each i = 0, 1, 2, \ldots . Termination can be characterized by a chain of dependency pairs.

Proposition 2.3: [1] A specification is terminating if and only if there is no infinite dependency chain.

DP(NAT+) = \{(+n(M, sN), +n(M, N))\} whose dependency pair is made from the second equation M + s N

\footnote{A substitution θ is a map from X to T. For a term t, the term whose all variable x are replaced with \theta(x) is denoted by \theta t. For example, let θ(0) = 0 and \theta(\#) = s. Then, (M + 0 N)θ = 0 + s N.}
chain: \[ s(M + N) \]

The following proposition holds.

5) \( l^\# > u^\# \) for each \((l^\#, u^\#) \in DP(E)\)

where \( > \) is defined as \( a > b \iff a \geq b \wedge b \not\geq a \). A quasi-order \( \geq \) is weakly monotonic if for all \( i \in \{ \overline{n} \} \), \( s_i \geq t_i \) implies \( \forall f, f(s_i) \geq f(t_i) \).

Note that \( \Sigma \) and \( T \) do not include renamed operation symbols \( g^\# \). The point of the above proposition is that equations are allowed to be interpreted into equations because of \( = \subseteq \geq \).

Thus we can use a model of a specification for constructing a model for proving termination. For a given specification, we define a specification \( SP^\# \) importing \( SP \) such that a model of \( SP^\# \) can be used for proving termination of \( SP \).

**Definition 3.2:** Let \( SP \) be a specification. The DP-specification of \( SP \), denoted by \( SP^\# \), is defined as the loose module including the followings:

1) a protected import of \( SP \),
2) a sort \( DP \), which is not included in \( SP \),
3) an operation symbol \( op \_\_\_\_ : DP \to Bool \),
4) an operation symbol \( op ^\# : \_\_\_\_ \to DP \)
5) an equation \( ep^\# \) for each dependency pair \((l^\#, u^\#) \in DP(SP)\).

Let \( SP = ((S, \leq, \Sigma), E) \) be a specification. A DP-specification \( SP^\# \) is formed as follows:

```plaintext
mod* SP# {
  pr(SP) {
    [DP]
    op \_\_\_\_ : DP \to Bool
    op f1# : \_\_\_\_ \to DP
    op f2# : \_\_\_\_ \to DP
    ... 
    ep f1#(...) > f1#(...) = true .
    ep f2#(...) > f1#(...) = true .
    ep f2#(...) > f2#(...) = true .
    ... 
  }
}
```

where \( pr(SP) \) stands for the import of \( SP \) with the protect mode. The arguments of \( f^\# \) (omitted as \( \ldots \)) are same with the original \( f \). For each \( ep f^\#(\ldots) > f^\#(\ldots) = true \), there exists \( f^\#(\ldots), f^\#(\ldots) \in DP(SP) \), and for each \( f^\#(\ldots), f^\#(\ldots) \in DP(SP) \), \( SP^\# \) includes \( ep f^\#(\ldots) > f^\#(\ldots) = true \). Let us consider a model of \( SP^\# \).

The sort \( DP \) can be interpreted into an arbitrary set, and renamed operation symbols \( f^\# \) can also be arbitrary functions. Since \( Bool \) denotes the Boolean algebra \( B \) and \( true \) stands for \( true \) of \( B \), the operation symbol \( \_\_\_\_ \) is interpreted into a predicate which returns \( true \) for each dependency pair. Since the import of \( SP \) is protected, a model \( M' \) of \( SP^\# \) interprets all \( SP \)'s contents as same as \( M \in [[SP]] \), i.e., for each \( M' \in [[SP^\#]] \) there exists \( M \in [[SP]] \) such that \( M_s = M'_s \) for each \( s \in S \) and \( M_f = M'_f \) for each \( f \in \Sigma \).

A model \( M \in [[SP^\#]] \) is well-founded if \( M \) is well-founded. We have the following theorem.

**Theorem 3.3:** If there exists a well-founded model \( M \in [[SP^\#]] \), then \( SP \) is terminating.

---

\*Each CafeOBJ specification implicitly imports the built-in module \( Bool \), which includes the sort \( Bool \), the constants \( true \) and \( false \) of \( Bool \), and the operation symbols \( and, or, etc \) of Boolean operations. \( Bool \) denotes the Boolean algebra \( B \), i.e. \( [[Bool]] = \{ B \} \), where \( B \) interprets elements in \( Bool \) naturally.

---

**III. PROVING TERMINATION BASED ON MODELS**

Existing methods and tools for proving termination, including those we introduced above, are used after describing specifications. Those methods and tools tell us "Yes", "No" or "Unknown" for the query of whether a specification is terminating or not, and they do not tell how to construct terminating ones. One way to give a guideline to describe terminating specification is to give a syntactic condition under which any specification satisfying the condition is terminating. However, it may restrict the flexibility of OBJ languages. Another way is to describe a specification with a terminating proof. To do it, specifiers are expected to be familiar with a termination method. Our motivation is to propose a termination method which is easy to be learned and used by OBJ users.

Since algebraic specifications denote \( \Sigma \)-algebras, semantic methods based on well-founded \( \Sigma \)-algebras seemingly may fit for algebraic specifications. However, as we showed above, they are completely different models. Equations \( l = r \) are interpreted into the equalities \( M_l = M_r \) in their denotational models, while they are interpreted into the inequalities \( M_l > M_r \) in well-founded \( \Sigma \)-algebras for termination, like the above example of polynomial interpretations for \( \text{NAT}^+ \), where \( M(x, y) = x + 2y + 1 \).

The DP method gives us a solution for reducing the gap between denotational models of specifications and well-founded models for termination. In classical methods, a rewrite relation corresponds to a well-founded order directly. Thus, each rewrite relation should be strictly decreased. In the DP method, only dependency pairs are strictly decreased, and rewrite relations are not needed to be strictly decreased. The following proposition holds.

**Proposition 3.1:** [1] A specification is terminating if there exists a quasi-order \( \geq \) on terms which satisfies the following conditions:

1) \( > \) is well-founded,
2) \( \geq \) is weakly monotonic,
3) \( \geq \) and \( > \) are closed under substation,
4) \( l \geq r \) for each \( l, r \in E \) and
5) \( l^\# > u^\# \) for each \((l^\#, u^\#) \in DP(E)\)

where \( > \) is defined as \( a > b \iff a \geq b \wedge b \not\geq a \). A quasi-order \( \geq \) is weakly monotonic if for all \( i \in \{ \overline{n} \} \), \( s_i \geq t_i \) implies \( \forall f, f(s_i) \geq f(t_i) \).
Proof of Theorem 3.3 Let $SP = (S, \leq, \Sigma, E)$. We give a $S$-sorted quasi-order $\geq$ satisfying Proposition 3.1. Define $\geq$ as follows:

a) For each $s \in S$ and $t, t' \in T_s$, $t \geq t'$ if $M_t = M_{t'}$.

b) For each $t, t' \in T_{sp}$, $t \geq t'$ if $\forall a. M_a(a(t), a(t'))$.

Note that $t$ and $t'$ in the case a) are terms of the original $SP$, i.e., no renamed symbols are included, and their sort are not DP. Consider the strict part $>_\geq$ of $\geq$. For each $t, t' \in T_s$ ($s \in S$), since $t \geq t' \Rightarrow M_t = M_{t'} \Rightarrow M_t = M_{t'}$, we obtain $t \geq t'$ from the definition of the case a), $t > t'$ does not hold. For each $t, t' \in T_{sp}$, if $t \geq t'$ then $t' \geq t$ from the well-foundedness of $M_{sp}$, therefore $>_\geq$ holds. Thus, we have the following lemma: $t > t'$ if and only if $\forall a. M_a(a(t), a(t'))$ for each $t, t' \in T_{sp}$.

We show $\geq$ satisfies 1) - 5) of Proposition 3.1.

1) Since $M_s$ are well-founded, $>_\geq$ is well-founded.

2) Assume $\forall i \in \{\bar{n}\}, s_i \geq t_i$. We show $f(s_n) \geq f(t_n)$ for each operation symbol $f$. The sorts of $s_i, t_i$ are not DP since no operation symbol can take terms of DP in its argument. From a), $M_{s_i} = M_{t_i}$, and thus $M_f(s_n) = M_f(t_n)$. From a) again, we have $f(s_n) \geq f(t_n)$.

3) Let $\theta$ be an arbitrary substitution and $t, t' \in T_s$ for some $s \in S$. Assume $t \geq t'$. From a) $M_t = M_{t'}$, and it means $\forall a. a(t) = a(t')$. Let $a_0$ be an arbitrary assignment. Define the assignment $a_0' = a_0(x) = a_0(\theta(x))$. Then, $a_0(\theta(t)) = a_0'(t)$. From the assumption, $a_0'(t) = a_0'(t')$. Thus, $\forall a. a(t) = a'(t') \Rightarrow a_0(t) = a_0'(t') \Rightarrow a_0(t') = a_0'(t') \Rightarrow a(t) = a(t')$ holds, and $>_\geq$ is closed under substitution $\theta$. For $>_\geq$, it can be proved similarly.

4) Since $M$ satisfies $l = r \in E$, i.e., $M_t = M_r$. Thus, $l > r$.

5) Let $(l^\#, u^\#) \in DP(SP)$. From the definition of $SP^\#$, $M_{l^\#}(a(l^\#), a(u^\#))$ holds for each assignment $a$. From the lemma, $l^\# > u^\#$. …

IV. DESCRIBING TERMINATING SPECIFICATIONS

In this section, we show examples of proving termination based on Theorem 3.3.

A. The specification NAT+

Since $DF(NAT+) = \{+\}$ and $DP(NAT+) = \{+\}$, the DP-specification NAT+ is given as follows:

```plaintext
mod* NAT+(#
  pr(NAT+)
  [DP]
  op _>_ : DP DP -> Bool
  op +# : Nat Nat -> DP
  var M N : Nat
  eq +#(M, s N) > +#(M, N) = true .
)
```

Note that the terms $+(M, s N)$ and $+(M, N)$ are of the sort DP. Let $N \in [NAT+]$. We give a model $M$ of NAT+ as follows: $M_x = N_x$ for all $x \in NAT+$ where $x$ is Nat, NatNat, Zero, 0, _+_, or _+. The sort DP is interpreted into the set of the natural numbers, i.e., $M_{sp} = N$. The operation symbol _+_, is interpreted into the ordinary order on $N$. The renamed operation symbol +# is interpreted as $M_{+#}(x, y) = y$. Then, $M$ satisfies the equations in NAT+.

B. The specification LIST

The following specification LIST specifies lists on natural numbers and the reverse function on the lists:

```plaintext
mod! LIST{
  pr(NAT+) [List]
  op nil : -> List
  op _-:_ : Nat List -> List
  pr revApp : List List -> List
  var L L' : List
  var N : Nat
  eq revApp(nil, L) = L .
  eq revApp(N ; L, L') = revApp(L, nil) .
  eq rev(L) = revApp(L, nil) .
}
```

The following is the CafeOBJ execution of applying the reverse function rev to the list [2,1,0], which returns the list [0,1,2]:

```
LIST> red rev(s 0 ; s 0 ; s 0 ; nil) .
... 0 ; (s 0 ; (s 0 ; nil)) : List
```

LIST denotes the following model $L$. All elements in NAT+ are interpreted as same as $N \in [NAT+]$. The sort List is interpreted into the set List($N$) of all lists on natural numbers. The operation symbols nil, _-_ and revApp are interpreted into the empty list [], the list constructor $L(x, [x_n]) = [x_n]$, the reverse function $rev : List(N) \rightarrow List(N)$, and the auxiliary function $L_{reverse}(l_0, l_1) = rev(l_0)@l_1$ where @ is the concatenation of lists. For example, $L_{reverse}([3,4],[2,1]) = [4,3,2,1]$.

We show termination of LIST. The DP-specification LIST# is given as follows:

```plaintext
mod* LIST#{
  pr LIST# [DP]
  op _>_ : DP DP -> Bool
  op +# : Nat Nat -> DP
  op rev# : List -> DP
  pr revApp# : List List -> DP
  var L L' : List
  var N : Nat
  eq +#(M, s N) > +#(M, N) = true .
  eq revApp#(N ; L, L') = revApp#(L, nil) .
```

The following is the CafeOBJ execution of applying the reverse function rev to the list [2,1,0], which returns the list [0,1,2]:

```
LIST> red rev(s 0 ; s 0 ; s 0 ; nil) .
... 0 ; (s 0 ; (s 0 ; nil)) : List
```

LIST denotes the following model $L$. All elements in NAT+ are interpreted as same as $N \in [NAT+]$. The sort List is interpreted into the set List($N$) of all lists on natural numbers. The operation symbols nil, _-_ and revApp are interpreted into the empty list [], the list constructor $L(x, [x_n]) = [x_n]$, the reverse function $rev : List(N) \rightarrow List(N)$, and the auxiliary function $L_{reverse}(l_0, l_1) = rev(l_0)@l_1$ where @ is the concatenation of lists. For example, $L_{reverse}([3,4],[2,1]) = [4,3,2,1]$.

We show termination of LIST. The DP-specification LIST# is given as follows:

```plaintext
mod* LIST#{
  pr LIST# [DP]
  op _>_ : DP DP -> Bool
  op +# : Nat Nat -> DP
  op rev# : List -> DP
  pr revApp# : List List -> DP
  var L L' : List
  var N : Nat
  eq +#(M, s N) > +#(M, N) = true .
  eq revApp#(N ; L, L') = revApp#(L, nil) .
```
The notion of dependency cycles has been proposed for analyzing which dependency pairs are essential to prove termination [1]. By using the dependency cycles, the dependency pair \((\text{rev}(\text{L}), \text{revApp}(\text{L}, \text{nil}))\) can be removed from elements of the dependency sequence. Thus, we do not need to give an interpretation for rev. See [1] for details.

V. CONCLUSION

We proposed a termination method for algebraic specifications based on their models. In our method, models of specifications are used for obtaining models for proving termination unlike classical semantic methods, e.g., polynomial interpretation. A combination of the dependency pair method and polynomial interpretation is known as a useful approach to give powerful termination prover [2], and is adopted by several termination provers like AProVe\(^{10}\), CiME\(^{11}\) and TTT\(^{12}\). Our method can be seen as a generalization of the combination of the dependency pair method and polynomial interpretation. In polynomial interpretation, all operation symbols are interpreted into a polynomials on natural numbers, where the order on natural numbers is well-founded. In our method, the sorts and the operation symbols in the original specification are interpreted by a denotational model of the specification, and the renamed operation symbols are interpreted into functions on a well-founded ordered set. Unfortunately, our method is not suitable for automated reasoning since it is impossible in practice to implement all models of algebraic specifications to be described in advance. Our method is useful for describing terminating specifications since a model of the specification exists on the mind of the specifiers. The purpose of a specifier is to describe specification of some model, for example, a system to be implemented.

In this article, we restrict target specifications to simple ones, that is, no operation attributes and no conditional equations, which are useful to describe practical specifications. For example, associative and commutative operation attributes are very useful to describe a collection type. The associativity \(((a \circ b) \circ c = a \circ (b \circ c))\) and the commutativity \((a \circ b = b \circ a)\) of a binary operation symbol \((\cdot \circ \cdot)\) are commonly used in algebraic specifications, however, they make a specification to be non-terminating and/or non-confluent. OBJ languages provide operation attributes to solve this problem. We describe associativity and commutativity as operation attributes instead of describing their equations directly. In reduction, those operation attributes are implemented in the matching phase, that is, the term \((a \circ b) \circ c\) is matched with the rewrite rule \((a \circ b) \circ a = d\) and is rewritten to \(d\) as a one-step rewriting. AC-TRS is a generalization of TRS, in which we declare associative and commutative operation symbols. Termination of AC-TRS has also been studied. Application of the dependency pair method to AC-TRS has been studied, for example, in [8], [7]. We may apply these results to obtain DP-specification with AC-symbols. A conditional equation is an equation equipped with a condition \(c\), which is a term of Boolean term, formed of \(l = r \iff c\). A model has to satisfy the body part \(l \theta = r \theta\) if the condition part \(c\) is interpreted into \(true\). In reduction, before applying the body part to a given term, the condition part is checked whether to be reduced into \(true\). CTRS is a generalization of TRS, in which we declare conditional rewrite rules. Besides infinite rewrite sequences we should care infinite calls of reductions of condition parts. Effective termination [12] and operational termination [9], [4] give the notion of termination for CTRS. Termination methods for effective (or operational) termination have been proposed. In the existing methods, effective (or operational) termination is proved by transforming a given CTRS into a TRS whose termination implies effective (or operational) termination of the original CTRS. Transformations may not preserve models of the original specification. To give a DP-specification for CTRS is one of our future work.

REFERENCES