

# A Sufficient Completeness Checker for Linear Order-Sorted Specifications Modulo Axioms<sup>\*</sup>

Joe Hendrix<sup>1</sup>, José Meseguer<sup>1</sup>, and Hitoshi Ohsaki<sup>2</sup>

<sup>1</sup> University of Illinois at Urbana-Champaign  
{jhendrix,meseguer}@uiuc.edu

<sup>2</sup> National Institute of Advanced Industrial Science and Technology  
ohsaki@ni.aist.go.jp

**Abstract.** We present a tool for checking the sufficient completeness of left-linear, order-sorted equational specifications modulo associativity, commutativity, and identity. Our tool treats this problem as an equational tree automata decision problem using the tree automata library CETA, which we also introduce in this paper. CETA implements a semi-algorithm for checking the emptiness of a class of tree automata that are closed under Boolean operations and an equational theory containing associativity, commutativity and identity axioms. Though sufficient completeness for this class of specifications is undecidable in general, our tool is a decision procedure for subcases known to be decidable, and has specialized techniques that are effective in practice for the general case.

## 1 Introduction

An equational specification is sufficiently complete when enough equations have been specified so that the functions defined by the specification are fully defined on all relevant data elements. This is an important property to check, both to debug and formally reason about specifications and equational programs. For example, many inductive theorem proving techniques are based on the constructors building up the data and require that the specification is sufficiently complete.

Sufficient completeness was introduced in the Ph.D. thesis of Guttag. (see [? ] for a more accessible treatment). For a good review of literature up to the late 80s, as well as some key decidability/undecidability results see [? ? ]. More recent developments show sufficient completeness as a tree automata decision problem (see [? ] and references there). For unsorted, unconditional, weakly-normalizing, and confluent specifications, the problem is EXPTIME-complete [? ].

Over the last 20 years, there have been numerous rewriting-based programming languages developed which support increasingly more expressive equational logics, including OBJ, Maude, ELAN, and CafeOBJ. These developments lead to a corresponding demand for reasoning tools that support these formalisms. In particular, there is a practical need for sufficient completeness checkers supporting: (1) conditional rewrite rules; (2) more expressive type formalisms such as order-sorted logic and membership equational logic; and (3) rewriting modulo associativity, commutativity, and identity. Our earlier work in [? ] addresses

---

<sup>\*</sup> Research supported by ONR Grant N00014-02-1-0715.

(1) and (2) through integration with an inductive theorem prover. Other recent work in [?] also addresses (1) using tree automata with constraints. The new tool we present in this paper addresses (2) and (3). Our checker is publicly available for download along with documentation and examples at the website: <http://maude.cs.uiuc.edu/tools/scc/>.

In an equational specification  $\mathcal{E} = (\Sigma, E)$  with rewriting modulo axioms, the equations are partitioned into two disjoint sets  $A$  and  $R$ . The set  $A$  consists of any combination of associativity, commutativity, and identity axioms. The other equations  $l = r \in R$  are treated as rewrite rules  $l \rightarrow r$  modulo  $A$ . A term  $t$  rewrites to  $u$  modulo  $A$ , denoted  $t \rightarrow_{R/A} u$  when there is a context  $C$  and substitution  $\theta$  such that  $t =_A C[l\theta]$  and  $C[r\theta] =_A u$ .

Our checker casts the left-linear, order-sorted sufficient completeness problem with rewriting modulo  $A$  as a decision problem for *equational tree automata* [?]. Equational tree automata over left-linear theories recognize precisely the equational closure of regular tree languages. However, since equational tree automata with associative symbols are not closed under Boolean operations [?], for checking properties such as inclusion, universality, and intersection-emptiness, we found it useful to introduce a new tree automata framework in [?], called *Propositional Tree Automata* (PTA), that is closed with respect both to Boolean operations and an equational theory.

## 2 Order-Sorted Sufficient Completeness

The motivation for sufficient completeness of a specification stems from the idea that introducing a new defined function should leave the underlying data elements unchanged. From a model-theoretic perspective, the initial model of the specification with the defined functions should be isomorphic to the initial model with only the constructor declarations. In the order-sorted context, we want to preserve this view of sufficient completeness, but the picture becomes more subtle due to the subsort relation and overloading — a symbol may be overloaded so that it is a constructor on one domain, and a defined symbol on another domain. As an example, we present a specification of lists of natural numbers with an associative append operator in Maude syntax.

```
fmod NATLIST is
  sorts Nat List NeList .      subsorts Nat < NeList < List .
  op 0 : -> Nat [ctor].      op s : Nat -> Nat [ctor].
  op nil : -> List [ctor].
  op _ : NeList NeList -> NeList [ctor assoc id: nil].
  op _ : List List -> List [assoc id: nil].
  op head : NeList -> Nat .   op tail : NeList -> List .
  op reverse : List -> List .
  var N : Nat .   var L : List .
  eq head(N L) = N .   eq tail(N L) = L .
  eq reverse(N L) = reverse(L) N .   eq reverse(nil) = nil .
endfm
```

In this specification, the signature  $\Sigma$  is defined by the `sort`, `subsort`, and operator declarations. The `ctor` attribute specifies an operator as a construc-

tor. The operator attributes `assoc` and `id: nil` define the axioms in  $A$ . The equations declarations define the rules in  $R$ . The append operator `__` is overloaded: it is defined on all lists, but only a constructor on non-empty lists.

In the unsorted context, sufficient completeness for weakly-normalizing and confluent specifications is usually checked by showing that every term containing a defined symbol at the root is reducible. In an order-sorted context in which the same symbol can be both a constructor and defined symbol, this check is *too strong*. Instead, we need to check that for each term  $f(t_1, \dots, t_n)$  where  $f : s_1 \dots s_n \rightarrow s$  is a defined symbol and every  $t_i$  is a constructor term of sort  $s_i$ , either  $f(t_1, \dots, t_n)$  is reducible, or  $f(t_1, \dots, t_n)$  is itself a constructor term of sort  $s$ . For details on why this property implies sufficient completeness, see [?] (which shows this in an even more general membership-equational context). It should be noted that there is an additional requirement for order-sorted specifications: the equations should be *sort-decreasing*. By this we mean that applying an equation  $l = r$  to a term  $l\theta$  of sort  $s$  should yield a term  $r\theta$  whose sort is less than or equal to  $s$ .

Our paper [?] shows in detail how to convert the sufficient completeness property into a propositional tree automata emptiness problem. The key idea is that given an order-sorted specification  $\mathcal{E} = (\Sigma, A \cup R)$  with sorts  $S$ , we can construct the following automata for each sort  $s \in S$ : (1) an automaton  $\mathcal{A}_s^c$  accepting constructor terms of sort  $s$ ; (2) an automaton  $\mathcal{A}_s^d$  accepting terms whose root is a defined symbol of sort  $s$  and whose subterms are constructor terms; and (3) an automaton  $\mathcal{A}^r$  accepting any term reducible by equations in  $R$ . If  $\mathcal{E}$  is weakly-normalizing, ground confluent, and ground sort-decreasing modulo  $A$ , then  $\mathcal{E}$  is sufficiently complete iff  $\mathcal{L}(\mathcal{A}_s^d) \subseteq \mathcal{L}(\mathcal{A}^r) \cup \mathcal{L}(\mathcal{A}_s^c)$  for each sort  $s \in S$ . Using our propositional tree automata framework, we in turn translate this problem into checking the emptiness of  $\bigcup_{s \in S} \mathcal{L}(\mathcal{A}_s^d) - (\mathcal{L}(\mathcal{A}^r) \cup \mathcal{L}(\mathcal{A}_s^c))$ .

This emptiness problem is decidable when the axioms in the specification are any combination of associativity, commutativity, and identity, except when a symbol is associative but not commutative. For the case of commutativity alone, this was shown in [?]. For symbols that are both associative and commutative, this was shown in [?]. Identity equations are transformed into identity rewrite rules using a specialized completion procedure along the lines of coherence completion in [?]. Our emptiness test identifies terms that are in normal form with respect to the identity rewrite rules. For symbols that are associative and not commutative, the emptiness problem is undecidable. For these symbols, our tool uses the semi-algorithm in [?], which we have found works well in practice. Collectively, these results allow our tool to handle specifications with any combination of associativity, commutativity, and identity axioms.

### 3 The Sufficient Completeness Checker (SCC)

The SCC has two major components: an analyzer written in Maude that generates the tree automaton emptiness problem from a Maude specification; and a C++ library called CETA that performs the emptiness check.

**Analyzer:** The analyzer accepts commands from the user, generates PTA from Maude specifications, forwards the PTA decision problems to CETA, and presents the user with the results. If the specification is not sufficiently complete, the tool shows the user a counterexample illustrating the error. The analyzer consists of approximately 900 lines of Maude code, and exploits Maude's support for reflection. The specifications it checks are also written in Maude.

If the user asks the tool to check the sufficient completeness of a specification that is not left-linear and unconditional, the tool transforms the specification by renaming variables and dropping conditions in to a checkable order-sorted left-linear specification. Even if the tool is able to verify the sufficient completeness of the transformed specification, it warns the user that it cannot show the sufficient completeness of the original specification. However, any counterexamples found in the transformed specification are also counterexamples in the original specification. We have found this feature quite useful to identify errors in Maude specifications falling outside the decidable class — including the sufficient completeness checker itself.

**CETA:** The propositional tree automaton generated by the analyzer is forwarded to the tree automata library CETA which we have developed. CETA is a complex C++ library with approximately 10 thousand lines of code. Emptiness checking is performed by a subset construction algorithm extended with support for associative and commutativity axioms as described in [? ]. The reason that CETA is so large is that the subset construction algorithm relies on quite complex algorithms on context free grammars, semilinear sets, and finite automata. We have found that CETA performs quite well for our purposes. Most examples can be verified in seconds. The slowest specification that we have checked is the sufficient completeness analyzer itself — the library requires just under a minute on a Pentium 4 desktop to check the 900 lines of Maude code in the analyzer. As an example, we present a tool session in which we check two specifications: NATLIST from the previous section; and NATLIST-ERROR which updates NATLIST to change the operator declaration of head from `op head : NeList -> Nat` to `op head : List -> Nat`.

```
Maude> in natlist.maude
=====
fmod NATLIST
=====
fmod NATLIST-ERROR
Maude> load scc.maude
Maude> loop init-scc .
Starting the Maude Sufficient Completeness Checker.
Maude> (scc NATLIST .)
Checking sufficient completeness of NATLIST ...
Success: NATLIST is sufficiently complete under the assumption that it is
      weakly-normalizing, ground confluent, and sort-decreasing.
Maude> (scc NATLIST-ERROR .)
Checking sufficient completeness of NATLIST-ERROR ...
Failure: The term head(nil) is a counterexample as it is an irreducible
      term with sort Nat in NATLIST-ERROR that does not have sort Nat in
      the constructor subsignature.
```

## 4 Conclusions

Our work in developing sufficient completeness checkers for more complex equational specifications has already led to two complementary approaches, each able to handle specifications outside classes that could be handled by previous approaches. Although significant progress has been made, there is a great deal of opportunity both to develop new techniques and to improve the performance of existing techniques. Additionally, the tools and techniques we have developed are not restricted to sufficient completeness. Recently, the CETA library has been integrated into the reachability analysis tool ACTAS [? ]. For more details on this, see CETA's website at: <http://formal.cs.uiuc.edu/ceta/>.