A Sufficient Completeness Reasoning Tool for Partial Specifications*

Joe Hendrix[1], Manuel Clavel[2], and José Meseguer[1]

[1] University of Illinois at Urbana-Champaign, USA
[2] Universidad Complutense de Madrid, Spain

Abstract. We present the Maude sufficient completeness tool, which explicitly supports sufficient completeness reasoning for partial conditional specifications having sorts and subsorts and with domains of functions defined by conditional memberships. Our tool consists of two main components: (i) a sufficient completeness analyzer that generates a set of proof obligations which if discharged, ensures sufficient completeness; and (ii) Maude's inductive theorem prover (ITP) that is used as a backend to try to automatically discharge those proof obligations.

1 Introduction

In computer science practice, equational specifications are often *partial.* That is, some of the relevant operations are only defined on an adequate subset of data. Simple examples of undefinedness include computing the top of an empty stack, division by zero, and many operations on data structures. This has led to the design of increasingly more expressive equational formalisms to deal with partiality (see [1] for a survey). In particular, the papers [1–3] proposed membership equational logic (MEL) as a framework logic for the equational specification of partial functions. The key idea is that the domain of definition of a partial function is axiomatized by conditional membership axioms stating when the function is defined. However, since conditional memberships may have arbitrarily complex conditions and equations may be conditional, in this setting the *sufficient completeness problem* is undecidable in general.

The Maude sufficient completeness tool (SCC), which analyzes MEL theories specified in Maude, is therefore not a decision procedure. Instead it is a reasoning tool consisting of two main components: (i) a *sufficient completeness analyzer* that generates a set of *proof obligations* which if discharged, ensures sufficient completeness of confluent, sort decreasing and reductive specifications; (ii) Maude's *inductive theorem prover* (ITP), that is used as a backend to try to automatically discharge those proof obligations.

Our tool has a number of useful applications. Two obvious ones are: (i) checking that the defined functions of a MEL specification will always evaluate to data

* Research supported by Grants ONR N00014-02-1-0715, NSF CCR-0234524, and Spanish MCYT Projects TIC2002-01167 and TIC2003-01000.

© Springer-Verlag Berlin Heidelberg 2005

built with the constructors; and (ii) for inductive theorem proving purposes, ensuring the correctness of the chosen proof technique (e.g. structural induction, cover set induction, inductionless induction, etc.) which typically depends on sufficient completeness. There are two other applications for which our tool has proved useful: (iii) checking that a rewrite theory specifying a concurrent system is *deadlock-free*, which is needed for verifying temporal logic properties using abstraction techniques [4]; the point is that deadlock-freeness can be characterized as the sufficient completeness of an associated MEL specification; and (iv) supporting more powerful *cover set induction schemes* in the style of [5] that can prove general conjectures of the form $\varphi(f(t_1,\ldots,t_n))$, where φ is a formula containing the expression $f(t_1,\ldots,t_n)$ with f a defined function symbol and the $t_1,\ldots,t_n$ constructor terms; the point here is that the sufficient completeness checker can be used to generate base cases in the induction scheme which are needed because in general the $t_1,\ldots,t_n$ may be nonvariable terms. This last application is a "turning of the tables" in the interoperation between Maude's ITP and SCC: in the second tool, the ITP plays an auxiliary role in discharging proof obligations, whereas in the ITP itself (which supports cover set induction) the SCC plays an auxiliary role in generating induction schemes.

2 Preliminaries

A MEL *signature* Σ is a triple $\Sigma = (\mathcal{K}, \Sigma, \mathcal{S})$, where $\mathcal{K}$ is a set of *kinds*, $\mathcal{S}$ is a disjoint $\mathcal{K}$-kinded family $\mathcal{S} = \{\mathcal{S}_k\}_{k\in\mathcal{K}}$ of sets of *sorts*, and $\Sigma = \{\Sigma_{w,s}\}_{(w,s)\in\mathcal{K}^*\times\mathcal{K}}$ is a $\mathcal{K}$-kinded signature of function symbols. Given a $\mathcal{K}$-kinded disjoint family of finite sets of variables $\vec{x} = x_1 : k_1, \ldots, x_n : k_n$, where $x_1, \ldots x_n$ are disjoint from the constants in Σ and the kinds $k_1, \ldots k_n$ in the list can be repeated, a *Σ-equation* is a formula $t = t'$, with $t, t' \in T_\Sigma(\vec{x})$, $T_\Sigma(\vec{x})$ being the free Σ-algebra on the variables $\vec{x}$, and such that t, t' have the same kind, i.e. $t, t' \in T_\Sigma(\vec{x})_k$ for some $k \in \mathcal{K}$. A *Σ-membership* is a formula $t : s$ such that if $t \in T_\Sigma(\vec{x})_k$, then $s \in \mathcal{S}_k$. *Σ-sentences* are universally quantified Horn clauses of the form

$$(\forall\vec{x})\ A\ \textbf{if}\ A_1 \wedge \cdots \wedge A_n$$

where A and the A_i are either Σ-equations or Σ-memberships. If A is a Σ-equation, we call the sentence a *conditional equation*; and if A is a Σ-membership, we call it a *conditional membership.* A MEL theory is a pair $\mathcal{E} = (\Sigma, \Gamma)$ with Σ a MEL signature and Γ a set of Σ-sentences. A *model* of a MEL signature $(\mathcal{K}, \Sigma, \mathcal{S})$ is a $(\mathcal{K}, \Sigma)$-algebra $\mathcal{A}$ together with a subset $\mathcal{A}_s \subseteq \mathcal{A}_k$, for each sort $s \in \mathcal{S}_k$. Then, models of a MEL theory $\mathcal{E} = (\Sigma, \Gamma)$ are models of Σ satisfying the axioms Γ. There is a sound and complete inference system to derive all theorems of a MEL theory (Σ, Γ) [1]. We denote the initial algebra of $\mathcal{E} = (\Sigma, \Gamma)$ by $T_\mathcal{E}$. There is a unique Σ-homomorphism $h : T_\mathcal{E} \to A$ for every model A of $\mathcal{E}$.

Under appropriate assumptions on the MEL theory $\mathcal{E}$ the conditional equations can be used from left to right as *rewrite rules* [2]. This is the way in which MEL is efficiently implemented in the Maude language [6]. An inference system for MEL reasoning is described in detail in Figure 7, page 57 of [2]. The notions of confluence and termination of term rewriting can be generalized to conditional MEL theories by corresponding notions of *confluence* and *reductiveness* [2]. Since sort computations are involved, a third important notion is *sort decreasingness.* Assuming that $\mathcal{E}$ is confluent and reductive, we can characterize sort decreasingness as the property that for each term t if we can infer $\mathcal{E} \vdash t : s$ with the rewrite inference system in Figure 7 of [2], then we can also infer $\mathcal{E} \vdash \mathrm{can}_{\mathcal{E}}(t) : s$ where $\mathrm{can}_{\mathcal{E}}(t)$ denotes the canonical form of t obtained by applying the confluent and reductive rewrite rules in $\mathcal{E}$. Intuitively, the more we simplify a term with the equations, the easier it becomes to compute its sort without having to remember any intermediate terms in the rewrite computation.

3 A Partial Specification Example

In Misra's data type of *powerlists* [7], a powerlist must be of length 2^n for some $n \in \mathbb{N}$, and the *zip* operator $\bowtie$ is only fully defined on powerlists of equal length. We can specify powerlists in MEL as a Maude functional module as follows:

```
fmod POWERLIST is protecting NAT . sort Pow .
  op [_] : Nat -> Pow [ctor] . ops _|_ _X_ : [Pow] [Pow] -> [Pow] .
  op len : Pow -> Nat .
  vars I J : Nat .             vars P Q R S : Pow .
  cmb (P | Q) : Pow if len(P) = len(Q) .
  cmb (P X Q) : Pow if len(P) = len(Q) [metadata "dfn"].
  eq [I] X [J] = [I] | [J] .   eq (P | Q) X (R | S) = (P X R) | (Q X S) .
  eq len([I]) = 1 .            eq len(P | Q) = len(P) + len(Q) .
endfm
```

The functional module `POWERLIST`includes the predefined module `NAT`, which declares the natural numbers with the expected arithmetic operations and relations. In the sort declaration section we introduce the sort `Pow`, which we will reserve for those terms representing powerlists; Maude automatically introduces also the kind `[Pow]` to denote the kind of the sort `Pow`. In the operator declaration section we introduce four operators: `[_]` for representing the operation that forms powerlist elements; `_|_` for representing the powerlist *tie* operation; `_X_` for representing the powerlist *zip* operation; and `len` for representing the operation that computes the length of a powerlist. Since we know that not all terms constructed with the operators `_|_` and `_X_` will represent powerlists, we declare those operators at the kind level. For example, $[4] \bowtie ([2] \mid [3])$ is not a powerlist. This is represented in `POWERLIST` by the fact that the term `[4] X ([2] | [3])` has kind `[Pow]`, but it does not belong to the sort `Pow`. On the other hand, since we want to use the `[_]` operator to *construct* powerlists (in particular, powerlists with only one element), we declare this operator at the sort level and with the `ctor` attribute. Finally, since we expect that the `len` operator applied to a powerlist will always evaluate to a natural number, we declare this operator at the sort level, but without the `ctor` attribute.

In the variable declaration section, we associate to the variables `I` and `J` the sort `Nat`, and to the variables `P`, `Q`, `R`, and `S` the sort `Pow`. By doing this, we are in fact declaring: i) that `I` and `J` are variables of the kind `[Nat]`, and `P`, `Q`, `R`, and `S` of the kind `[Pow]`, and ii) that in all memberships and equations in which those variables appear, there is an extra condition stating that those variables only range over the set of terms belonging to their associated sort. Finally, in the membership declaration section, we declare that both the *tie* and the *zip* of two powerlists are powerlists if they have equal length; however, since we do not want to use the `_X_` operator as a *constructor* operator for terms representing powerlists, but rather as a *defined* operator, we declare the membership for the `_X_` operator with the `dfn` attribute. In fact, if we go back to the operator declarations section, we can realize that

```
op [_] : Nat -> Pow [ctor] .  op len : Pow -> Nat .
```

is just syntactic sugar for the following declarations:

```
op [_] : [Nat] -> [Pow] .     op len : [Pow] -> [Nat] .
mb [I]: Pow .                 mb len(P): Nat  [metadata "dfn"].
```

As we will explain in the following section, the sufficient completeness problem for `POWERLIST` reduces to proving that all terms P `X` Q and `len(`P`)`, where P and Q are terms built with our *constructor* memberships, can be proved to be of sort `Pow` without using the *defined* memberships.

4 Sufficient Completeness for MEL Specifications

The definition of sufficient completeness for MEL specifications is somewhat subtle, in that in its most general form it cannot be given only in terms of a subsignature Ω of constructors. The point is that, when specifying the conditional memberships for constructor operators in Ω, other nonconstructor function symbols may appear in the condition. This is illustrated in the powerlist example by the conditional membership for the constructor $\_|\_$ of powerlists. The definition below strictly generalizes that in [2], which ruled out the use of nonconstructor symbols in conditions of constructor memberships.

Definition 1. *Let $\mathcal{E} = ((\mathcal{K}, \Sigma, \mathcal{S}), E \cup M_< \cup M_\Sigma)$ be a MEL specification where E contains the conditional equations, $M_<$ contains the memberships corresponding to subsort declarations explained below, and M_Σ contains the conditional memberships specifying the sorts of function symbols in Σ. Subsort declarations $s < s'$ with $s \neq s'$ and $s, s' \in S_k$ for some k are axiomatized by the conditional membership:*

$$(\forall x : k)\ x : s' \textbf{ if } x : s$$

Finally, we assume that any conditional membership in M_Σ is of the form:

$$(\forall \vec{x})\ f(t_1, \ldots, t_n) : s \textbf{ if } t_1 : s_1 \wedge \cdots \wedge t_n : s_n \wedge \mathcal{C} \qquad (1)$$

where $f \in \Sigma$, $\vec{x} = \mathrm{var}(f(t_1, \ldots, f_n))$, and $\mathcal{C}$ is a (possibly empty) conjunction of Σ-equations and Σ-memberships, $\mathrm{var}(\mathcal{C}) \subseteq \vec{x}$, and if f is a constant in $\Sigma_{\epsilon,k}$ then $\mathcal{C}$ is empty.

Given a subset of memberships $M_\Omega \subseteq M_\Sigma$, called constructor memberships, *we define a* constructor subtheory *to be $\mathcal{E}_\Omega = ((\mathcal{K}, \Sigma, \mathcal{S}), E \cup M_< \cup M_\Omega)$. Furthermore, we say that $\mathcal{E}$ is* sufficiently complete *relative to M_Ω iff $\mathcal{E}_\Omega$ is such that the unique Σ-homomorphism*

$$h : T_{\mathcal{E}_\Omega} \to T_{\mathcal{E}}$$

is an isomorphism. Finally, we define M_Δ to be $M_\Sigma - M_\Omega$.

To illustrate these notions, we can use (the desugared version of) `POWERLIST`. In this specification: M_Σ is the set containing the memberships

```
(1)  mb 0 : Nat .
(2)  cmb s N : Nat if N : Nat .
(3)  mb [I]: Pow .
(4)  cmb (P | Q) : Pow if len(P) = len(Q) .
(5)  mb len(P): Nat [metadata "dfn"].
(6)  cmb (P X Q) : Pow if len(P) = len(Q) [metadata "dfn"].;
```

$M_<$ is the empty set; M_Δ is the set containing (5) and (6), that is the memberships labeled with `dfn`; and M_Ω is the set containing (1)–(4).

The soundness of the Maude sufficient completeness tool is based on the following theorem, which we have proven in the technical report [8].

Theorem 1. *Let $\mathcal{E} = (\Sigma, E \cup M_< \cup M_\Omega \cup M_\Delta)$ be a MEL specification satisfying:*

(i) $\mathcal{E}$ and $\mathcal{E}_\Omega$ are reductive, ground confluent, and ground sort-decreasing.
(ii) Each membership in $M_\Omega \cup M_\Delta$ is of the restricted form (1).

Then the two statements below are equivalent:

(a) $\mathcal{E}$ is sufficiently complete relative to constructor memberships M_Ω
*(b) For each membership $(\forall \vec{x})\ t : s$ **if** $\mathcal{C}$ in M_Δ and ground substitution $\theta : \vec{x} \to T_\Sigma$ such that $\mathcal{E}_\Omega \models \mathcal{C}\theta$, either $t\theta$ is $\mathcal{E}_\Omega$-reducible or there is a membership in M_Ω of the form $(\forall \vec{y})\ u : s'$ **if** $\mathcal{C}'$ with $s' \leq s$ and a substitution $\tau : \vec{y} \to T_\Sigma$ such that $t\theta = u\tau$ and $\mathcal{E}_\Omega \models \mathcal{C}'\tau$.*

Due to space constraints, we do not reproduce the proof here: consult [8] for the detailed proof.

5 The Maude Sufficient Completeness Tool

The Maude Sufficient Completeness tool (SCC) is itself written in Maude using *reflection*. (More details on reflection in Maude in Sect. 5.2.) The soundness of the tool is based on Theorem 1. There are two major components to the tool: a *Sufficient Completeness Analyzer*, which generates proof obligations for sufficient completeness problems, and the Maude *Inductive Theorem Prover* (ITP), extended with additional commands to try to automatically prove those proof obligations. The tool has been run on a variety of different MEL specifications, and is available for download with source, documentation, and examples (including MEL specifications of ordered lists with sorting functions, stacks, and binary trees) from the tool's webpage: `http://maude.cs.uiuc.edu/tools/scc/`

5.1 The Maude Sufficient Completeness Analyzer

The Maude Sufficient Completeness Analyzer follows the incremental constructor-based narrowing of patterns approach, but generalized to handle conditional specifications. Given a MEL theory $\mathcal{E} = (\Sigma, E \cup M_< \cup M_\Sigma)$ in Maude, conveniently annotated to indicate a constructor subtheory $\mathcal{E}_\Omega$, the Maude sufficient completeness analyzer generates, in a two phase process, a set of proof obligations which if discharged, ensures the sufficient completeness of $\mathcal{E}$ relative to M_Ω. The sufficient completeness analyzer assumes that $\mathcal{E}$ satisfies the requirements (i) and (ii) in Theorem 1.

The Narrowing Procedure. In its first phase, the analyzer returns a set $\Delta = \{(t, s, \mathcal{C})_i\}_{i \in \mathbb{N}}$ such that, if t' is a counterexample for sufficient completeness, then there exists a triple $(t, s, \mathcal{C}) \in \Delta$ and a substitution $\theta : \mathrm{var}(t) \to T_\Sigma$ such that $t' = \theta t$ and $\mathcal{E}_\Omega \models \theta\mathcal{C}$. The set Δ is generated from the initial set $\{(t, s, C) \mid (\forall \vec{x})\ t : s \textbf{ if } \mathcal{C} \in M_\Delta\}$ by applying rule (2) below until it cannot be applied anymore. The rule (2) uses the *expandability relation* $\blacktriangleleft$ and the *expand function* exp which are defined as follows:

Definition 2. *Let t, t' be terms in $T_\Sigma(\vec{x})$ such that* $\mathrm{var}(t) \cap \mathrm{var}(t') = \emptyset$*, and $x \in \mathrm{var}(t)$. Then, $t \blacktriangleleft_x t'$ iff there exists a substitution θ such that θ is a most general unifier of t and t', and $\theta(x)$ is not a variable.*

Definition 3. *Let $t \in T_\Sigma(\vec{x})_k$, $s \in \mathcal{S}_k$, $\mathcal{C}$ a conjunction of atomic formulas, $x \in \vec{x}$ with $x : s' \in \mathcal{C}$, and M a set of memberships whose variables have all been renamed to be disjoint from $\vec{x}$. Then,*

$$\exp(t, s, \mathcal{C}, x, M) = \{(t\theta, s, \mathcal{C}\theta \wedge \mathcal{C}') \mid (\forall \vec{y})\ u : s' \textbf{ if } \mathcal{C}' \in M, \theta = (x \mapsto u)\}$$

Finally, we define the inference rule that generates the set Δ. Note that this rule will only be applied a finite number of times, because of the condition $t \blacktriangleleft_x t'$ on the rule.

Δ-rule For any $(\forall \vec{y})\ t' = t''$ **if** $\mathcal{C}'$ in E,

$$\frac{\Delta' \cup \{(t, s, \mathcal{C})\}}{\Delta' \cup \exp(t, s, \mathcal{C}, x, M_< \cup M_\Omega)} \quad \text{if } x \in \mathrm{var}(t), t \blacktriangleleft_x t' \tag{2}$$

The Proof Obligation Generator. In its second phase, the SCC produces, from the set Δ, a set of proof obligations which if discharged, guarantees that $\mathcal{E}$ is sufficiently complete with respect to M_Ω. Since a triple $(t, s, \mathcal{C}) \in \Delta$ represents a set of potential counterexamples, the proof obligation generator produces a sentence which if proven in $\mathcal{E}_\Omega$, implies that for every substitution $\theta : \mathrm{var}(t) \to T_\Sigma$ at least one of the following holds and, therefore by Theorem 1, that $\mathcal{E}$ is sufficiently complete with respect to $\mathcal{E}_\Omega$:

a) $\mathcal{E}_\Omega \not\models \mathcal{C}\theta$
b) $t\theta$ is *reducible*
c) There exists a membership $(\forall \vec{y})\ u : s'$ **if** $\mathcal{C}'$ in M_Ω with $s' < s$ and a substitution $\tau : \vec{y} \to T_\Sigma$ such that $t\theta = u\tau$ and $\mathcal{E}_\Omega \models \mathcal{C}'\tau$.

In particular, for each $(t, s, \mathcal{C}) \in \Delta$, the proof obligation generator constructs the sentence:

$$(\forall x \in \mathrm{var}(t)) \left(\neg\mathcal{C} \vee \bigvee_{\substack{t'=t'' \text{ if } \mathcal{C}' \in E, \\ \theta \text{ s.t. } \mathbb{C}[t'\theta]=t}} \mathcal{C}'\theta \quad \vee \bigvee_{\substack{u{:}s' \text{ if } \mathcal{C}' \in M_\Omega \text{ s.t. } s'<s, \\ \theta \text{ s.t. } u\theta = t}} \mathcal{C}'\theta \right) \tag{3}$$

where $\mathbb{C}$ denotes a context.

5.2 The Maude ITP

The ITP [9] tool is an experimental interactive tool for proving properties of MEL specifications in Maude. The ITP tool has been written entirely in Maude, and is in fact an *executable* specification in MEL of the formal inference system that it implements. The ITP inference system treats MEL specifications as *data* – for example, one inference may add to the specification an induction hypothesis as a new equational axiom. This makes a *reflective* design, in which Maude equational specifications become data at the metalevel, ideally suited for implementing the ITP. Using reflection to implement the ITP tool has one important additional advantage, namely, the ease to rapidly extend it by integrating other tools implemented in Maude using reflection, as it is the case of the SCC.

In the ITP, the user introduces commands which are interpreted as actions that may change the state of the proof, specifically the set of goals that remain to be proved, with each goal consisting of a formula to be proved and the MEL specification in which the formula must be proved. After executing the action requested by the user, the tool reports the resulting state of the proof. The main module implementing the ITP is the `ITP-TOOL` module. In this module, states of proofs, sets of goals, goals and formulas are represented by terms of different sorts, and the actions interpreting the ITP commands are represented as different, equationally defined functions over those terms.

To integrate the SCC in the ITP we have added two new commands, `scc` and `scc*`, to the ITP; the `scc*` command is an extension of `scc` that takes into account the information obtained by this command *at run-time*. We begin with the `scc` command. This command is implemented by extending the module `ITP-TOOL` with a new, equationally defined function that, given an equational specification $\mathcal{E}$, does the following:

- first, it calls on $\mathcal{E}$ the function `checkCompleteness`, which implements the sufficient completeness analyzer described in Sect. 5.1;
- then, it converts the resulting proof obligations into a set of ITP goals, which are all associated with $\mathcal{E}_\Omega$;
- finally, it eliminates from the state of the proof those goals that can be proved automatically using the ITP `auto*` command[1].

[1] The current implementation of the `auto*` command integrates its rewriting-based simplification strategy with a decision procedure for linear arithmetic with uninterpreted function symbols; this theory includes many of the formulas that one tends to encounter in proof obligations generated by the SCC tool.

As an example, we can use the `scc` command to check the sufficient completeness of `POWERLIST`. After introducing in Maude the command line `(scc POWERLIST .)`, the ITP tool reports the resulting state of the proof:

```
==================================
CTOR-POWERLIST$1.0
==================================
|- A{P:Pow ; Q':Pow}
    ((~(len(P:Pow)= len(Q':Pow)))V(~(len(P:Pow)+ len(Q':Pow)= 1)))
==================================
CTOR-POWERLIST$2.0
==================================
|- A{P:Pow ; Q:Pow}
    ((~(len(P:Pow)= len(Q:Pow)))V(~(len(Q:Pow)+ len(P:Pow)= 1)))
```

In this case, the `auto*` command failed to discharge the above goals corresponding to proof obligations generated by SCC, despite the fact that the formulas associated to those goals are obviously true in $\mathcal{E}_\Omega$ The reason is the following. In $\mathcal{E}_\Omega$, the `len` operator is declared at the kind level: it takes a term of the kind `[Pow]` and returns a term of the kind `[Nat]`. In this situation, the decision procedure cannot recognize the formulas associated to the above goals as belonging to the class of formulas that it can solve. Therefore, to discharge the proof obligations it is necessary to prove that the operator `len` always returns a term of sort `Nat` when it is called on terms of sort `Pow`. This is, however, implied by the fact that SCC has generated no proof obligations for the `len` operator.

Since the situation described above is a rather common one, we have also implemented the command `scc*` that associates all the goals corresponding to the proof obligations generated by SCC with $\mathcal{E}_\Omega$, but extended this time with all the operator declarations in $\mathcal{E}$ that SCC has found unproblematic. In the case of powerlists, `scc*` discharges all the proof obligations automatically.

6 Related Work

We cannot survey here the extensive literature on sufficient completeness: we just mention some related work to place things in context. Sufficient completeness of MEL specifications was first studied in [2]; the definition and methods on which the present tool is based are strictly more general than those in [2], allowing a much wider class of MEL specifications to be checked. Sufficient completeness itself goes back to Guttag's thesis [10] (but see [11] for a more formal, direct treatment of this notion). An early algorithm for handling unconditional linear specifications is due to Nipkow and Weikum [12]. For a good review of the literature up to the late 80s, as well as some important decidability/undecidability and complexity results, see [13, 14]. A more recent development is the casting of the decidable cases of sufficient completeness as tree automata decision problems: see Chapter 4 of [15] and references there. Two sufficient completeness tools having a similar approach to ours, namely the incremental constructor-based narrowing of patterns, are the sufficient completeness checkers of the Spike [16] and RRL [17] theorem provers, both of which are based on many-sorted equational logic. By contrast, our approach is based on a more expressive partial equational logic (MEL). However, RRL [17], although based on a total many-sorted logic, can address some partiality issues in a different way: incompleteness can be due to *omissions*, yielding real counterexample patterns, or can be *intentional*, due to partiality, in which case the partial function's domain of definition can be specified by a quantifier-free formula, which can be used to ascertain if a counterexample pattern is relevant in that domain.

7 Conclusions and Future Work

At present, the SCC can handle specifications where some symbols have been declared commutative. Future work will extend the tool to handle equations modulo different combinations of associativity, commutativity, and identity. It is well-known that sufficient completeness is undecidable in the presence of associative axioms, even for left-linear confluent and terminating equations [14]. However, equational tree automata techniques in the style of [18] can still make the problem decidable for some subclasses, and the ITP can support reasoning to discharge proof obligations for the general case.

As already mentioned, the tool assumes MEL specifications $\mathcal{E}$ that are ground confluent, reductive, and sort-decreasing. Although Maude already has tools to check these properties in the special case where $\mathcal{E}$ is an *order-sorted* specifications [19], tools to discharge the corresponding obligations for general MEL specifications need to be developed. For termination of MEL specifications there is already a tool prototype [20] and supporting theory [20, 21]. For checking confluence and sort-decreasingness of *general* MEL specifications detailed supporting theory can be found in [2], but a tool needs to be developed.

References

1. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: In 12th International Workshop on Recent Trends in Algebraic Development Techniques (WADT'97). Volume 1376 of Lecture Notes in Computer Science., Springer-Verlag (1998) 18–61
2. Bouhoula, A., Jouannaud, J.P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science **236** (2000) 35–132
3. Meseguer, J., Roşu, G.: A total approach to partial algebraic specification. In: Proceedings of ICALP. Volume 2380 of Lecture Notes in Computer Science., Springer (2002) 572–584
4. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstractions. In: Proceedings of CADE. Volume 2741 of Lecture Notes in Computer Science., Springer (2003) 2–16
5. Kapur, D., Subramaniam, M.: New uses of linear arithmetic in automated theorem proving by induction. Journal of Automated Reasoning **16** (1996) 39–78
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J.: Maude: Specification and programming in rewriting logic. Theoretical Computer Science **285** (2002) 187–243

7. Misra, J.: Powerlist: a structure for parallel recursion. ACM Transactions on Programming Languages and Systems **16** (1994) 1737–1767
8. Hendrix, J., Clavel, M., Meseguer, J.: A sufficient completeness reasoning tool for partial specifications (extended technical report). Available on tool website at `http://maude.cs.uiuc.edu/tools/scc/` (2005)
9. Clavel, M.: The ITP tool's home page. http://maude.sip.ucm.es/itp (2005)
10. Guttag, J.: The Specification and Application to Programming of Abstract Data Types. PhD thesis, University of Toronto (1975) Computer Science Department, Report CSRG-59.
11. Guttag, J.V., Horning, J.J.: The algebraic specification of abstract data types. Acta Inf. **10** (1978) 27–52
12. Nipkow, T., Weikum, G.: A decidability result about sufficient-completeness of axiomatically specified abstract data types. In: Proc. 6th GI-Conf. Theoretical Computer Science. Volume 145 of Lecture Notes in Computer Science., Springer (1983) 257–268
13. Kapur, D., Narendran, P., Zhang, H.: On sufficient-completeness and related properties of term rewriting systems. Acta Informatica **24** (1987) 395–415
14. Kapur, D., Narendran, P., Rosenkrantz, D.J., Zhang, H.: Sufficient-completeness, ground-reducibility and their complexity. Acta Informatica **28** (1991) 311–350
15. Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S., Tommasi, M.: Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata` (1997) release October, 1st 2002.
16. Bouhoula, A., Rusinowitch, M.: SPIKE: A system for automatic inductive proofs. In: Algebraic Methodology and Software Technology, AMAST '95, Proceedings. Volume 936 of Lecture Notes in Computer Science., Springer (1995) 576–577
17. Kapur, D.: An automated tool for analyzing completeness of equational specifications. In: Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA), August 17-19, 1994, Seattle, WA, USA. Software Engineering Notes, Special Issue, ACM Press (1994) 28–43
18. Ohsaki, H., Seki, H., Takai, T.: Recognizing boolean closed a-tree languages with membership conditional rewriting mechanism. In: Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings. Volume 2706 of Lecture Notes in Computer Science., Springer (2003) 483–498
19. Clavel, M., Durán, F., Eker, S., Meseguer, J.: Building equational proving tools by reflection in rewriting logic. In: Cafe: An Industrial-Strength Algebraic Formal Method. Elsevier (2000)
20. Durán, F., Lucas, S., Meseguer, J., Marché, C., Urbain, X.: Proving termination of membership equational programs. In: Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2004, Verona, Italy, August 24-25, 2004, ACM Press (2004) 147–158
21. Lucas, S., Meseguer, J., Marché, C.: Operational termination of generalized conditional term rewriting systems. Submitted. (2004)