its simplicity and the small amount of additional communication overhead that is required. An analysis of the system is described to determine its mean time to failure.

Acknowledgment

The author would like to thank G. Wuu and S. Chodra for comments and assistance in performing the calculations.

References

[1] P. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Comput. Surveys*, vol. 13, June 1981.
[2] J. Banino, C. Kaiser, and H. Zimmermann, "Synchronization for distributed systems," presented at the 1st Int. Conf. Distributed Comput. Syst., Huntsville, AL, Oct. 1979.
[3] D. Boggs, "Internet broadcasting," Ph.D. dissertation, Dep. Elec. Eng., Stanford Univ., Stanford, CA, Jan. 1982.
[4] H. Breitwieser and M. Leszak, "A distributed transaction processing protocol based on majority consensus," in *Proc. ACM SIGACT-SIGOPS Symp. Principles of Distributed Comput.*, Ottawa, Ont., Canada, Aug. 1982.
[5] F. Cristian, "A rigorous approach to fault tolerant programming," in *Proc. 4th Jerusalem Conf. Inform. Technol.*, Jerusalem, Israel, May 1984.
[6] D. Dolev, "The Byzantine generals strike again," *J. Algorithms*, vol. 3, 1982.
[7] D. Daniels and A. Spector, "An algorithm for replicated directories," presented at the 3rd ACM Symp. Principles of Distributed Comput., Montreal, P.Q., Canada, Aug. 1983.
[8] D. Gifford, "Weighted voting for replicated data," in *Proc. 7th Symp. Oper. Syst. Principles*, Pacific Grove, CA, Dec. 1979.
[9] E. Holler, "Multiple copy update," in *Lecture Notes in Computer Science, Distributed Systems–Architecture and Implementation*, vol. 105. Berlin, Germany: Springer-Verlag, 1981.
[10] J. Kemeny and J. Snell, *Finite Markov Chains*. Princeton, NJ: Van Nostrand, 1960.
[11] L. Lamport, "Time, clocks and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, July 1978.
[12] —, "Using time instead of timeout for fault-tolerant distributed systems," *ACM Trans. Program. Lang. Syst.*, vol. 6, Apr. 1984.
[13] B. Lampson, "Atomic transactions," in *Lecture Notes in Computer Science, Distributed Systems–Architecture and Implementation*, vol. 105. Berlin, Germany: Springer-Verlag, 1981.
[14] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, July 1982.
[15] A. Tannenbaum, *Computer Networks*. Englewood Cliff, NJ: Prentice-Hall, 1981.
[16] R. Thomas, "A solution to the concurrency control problem for multiple copy databases," presented at IEEE COMPCON '78, Apr. 1978.

Arthur J. Bernstein (S'56–M'63–SM'78–F'81), for a biography, see p. 67 of the January 1985 issue of this Transactions.

Completeness of Proof Systems for Equational Specifications

DAVID B. MACQUEEN AND DONALD T. SANNELLA

Abstract—Contrary to popular belief, equational logic with induction is not complete for initial models of equational specifications. Indeed, under some regimes (the Clear specification language and most other algebraic specification languages) no proof system exists which is complete even with respect to ground equations. A collection of known results is presented along with some new observations.

Index Terms—Algebraic specifications, equational logic, proof systems.

Manuscript received February 1, 1984; revised December 10, 1984. This work was supported by the Science and Engineering Research Council of Great Britain and AT&T Bell Laboratories.

D. B. MacQueen is with AT&T Bell Laboratories, Murray Hill, NJ 07974.
D. T. Sannella is with the Department of Computer Science, University of Edinburgh, Edinburgh EH9 3JZ, Scotland.

I. Introduction

Since the pioneering work of Guttag [18], Zilles [30], and the ADJ group [16] there has been substantial work on applying universal algebra to the specification of abstract data types and programs. Typical specifications involve several sorts (kinds of data), so standard universal algebra as in [17] is not immediately applicable since it deals with algebras having only a single sort. Heterogeneous or many-sorted universal algebra as developed in [5] and [20] is therefore more useful for this purpose. The generalization to many sorts has been widely assumed to be straightforward, but recently Goguen and Meseguer [14] have shown that a classical result of universal algebra, that equational logic is sound and complete for equational specifications, does not automatically extend from the one-sorted to the many-sorted case. This surprising fact

A signature Σ_{bool}

sorts bool
opns false, true : bool
  not : bool $\rightarrow$ bool
  and, or : bool,bool $\rightarrow$ bool

Some Σ-equations E_{bool}

$\forall\phi$. not(true) = false
$\forall\phi$. not(false) = true
$\forall$x:bool. and(x,x) = x
$\forall$x:bool. and(x,not(x)) = false
$\forall$x:bool. or(x,x) = x
$\forall$x:bool. or(x,not(x)) = true

A Σ_{bool}-algebra A (a model of Bool=$\langle\Sigma_{\mathrm{bool}},E_{\mathrm{bool}}\rangle$)

$|A|_{\mathrm{bool}} = \{0,1,2\}$

$\mathrm{false}_A = 0$
$\mathrm{true}_A = 1$

$\mathrm{not}_A(0) = 1$
$\mathrm{not}_A(1) = \mathrm{not}_A(2) = 0$

$\mathrm{and}_A(0,0) = 0$ $\mathrm{and}_A(0,1) = 0$ $\mathrm{and}_A(0,2) = 2$
$\mathrm{and}_A(1,0) = 0$ $\mathrm{and}_A(1,1) = 1$ $\mathrm{and}_A(1,2) = 2$
$\mathrm{and}_A(2,0) = 0$ $\mathrm{and}_A(2,1) = 2$ $\mathrm{and}_A(2,2) = 2$

$\mathrm{or}_A(0,0) = 0$ $\mathrm{or}_A(0,1) = 1$ $\mathrm{or}_A(0,2) = 2$
$\mathrm{or}_A(1,0) = 1$ $\mathrm{or}_A(1,1) = 1$ $\mathrm{or}_A(1,2) = 2$
$\mathrm{or}_A(2,0) = 1$ $\mathrm{or}_A(2,1) = 2$ $\mathrm{or}_A(2,2) = 2$

Fig. 1. Signatures, equations, algebras.

refutes certain results in [2] and [20]. Goguen and Meseguer then show how the usual rules of equational deduction can be modified to give a system which is sound and complete in the many-sorted case.

But this is not the whole story. Ordinary many-sorted equational algebra is not sufficiently powerful for use in specification because it provides no way of excluding certain undesirable models (for example, it is impossible to specify Booleans in such a way that true $\neq$ false is guaranteed). Accordingly, work on algebraic specification uses many-sorted algebra with extensions to provide the necessary power. First attempts (e.g., [16]) restricted attention to the *initial* models of a specification. Later work on parameterized and "loose" specifications (e.g., in the Clear specification language [8]; cf. [21]) generalized this idea, permitting equational specifications to include *data constraints* (or *hierarchy constraints* as in CIP-L [1]) specifying that certain subalgebras of any model must be *free extensions* of (respectively *generated* from) certain smaller subalgebras. It seems to have been assumed by many that equational logic with these modifications remains sound and complete if an appropriate induction rule is added. This assumption turns out to be false; such a proof system is indeed sound, but it is not complete. Indeed, no sound and complete proof system exists in either extended situation.

II. Preliminaries—Specifications, Models, Proof Systems, and Completeness

The following short sequence of definitions is a necessary prerequisite to the statement of completeness results for those who are not familiar with previous work on algebraic specification. The first five definitions are adapted from [8]. Some familiar examples of some of these concepts are given in Fig. 1.

Definition: A *signature* Σ is a set of *sorts* (data type names) together with a set of *operators* (operation names), where each operator has a *type* of the form $s1, \cdots, sn \rightarrow s$ where $s1, \cdots, sn$, s are sorts. If Σ and Σ' are signatures then Σ is a *subsignature* of Σ', written $\Sigma \subseteq \Sigma'$, if sorts(Σ) $\subseteq$ sorts(Σ') and operators(Σ) $\subseteq$ operators(Σ').

Definition: Given a signature Σ, a Σ-*algebra* A consists of a countable *carrier* set $|A|_s$ for each sort s of Σ (the possible values of that data type) together with a total function $\omega_A: |A|_{s1} \times \cdots \times |A|_{sn} \rightarrow |A|_s$ for each operator $\omega: s1, \cdots, sn \rightarrow s$ of Σ. If $\Sigma' \subseteq \Sigma$, then the *restriction* of A to Σ' (written $A|_{\Sigma'}$) is the Σ'-algebra obtained by forgetting the carriers and functions of A corresponding to sorts and operators which are not in Σ'.

Definition: Given a signature Σ and Σ-algebras A, A', a Σ-*homomorphism* $f: A \rightarrow A'$ consists of a map $f_s: |A|_s \rightarrow |A'|_s$ for each sort s of Σ which preserves the operations; i.e., for every operator $\omega: s1, \cdots, sn \rightarrow s$ in Σ and every $a_1 \in |A|_{s1}, \cdots, a_n \in |A|_{sn}$, $f_s(\omega_A(a_1, \cdots, a_n)) = \omega_{A'}(f_{s1}(a_1), \cdots, f_{sn}(a_n))$. If $\Sigma' \subseteq \Sigma$, then the *restriction* of f to Σ' (written $f|_{\Sigma'}$) is the Σ'-homomorphism $f|_{\Sigma'}: A|_{\Sigma'} \rightarrow A'|_{\Sigma'}$ obtained by restricting f to the sorts in Σ'. A homomorphism $f: A \rightarrow A'$ which is 1-1 and onto is called an *isomorphism*, written $A \cong A'$.

Definition: Given a signature Σ, a Σ-*equation* $\forall X. t = t'$ is a finite set X of variables of sorts in Σ together with a pair t, t' of Σ-terms (possibly containing variables from X) of the same sort. Terms and equations containing no variables are called *ground*. A Σ-algebra A *satisfies* a Σ-equation $\forall X. t = t'$ (written $A \models \forall X. t = t'$) if the equation is "true" (both sides evaluate to the same thing) for all assignments of values in A to the variables in X.

Definition: A *specification* is a signature Σ together with a set of Σ-equations. A Σ-algebra A *satisfies* a specification $\langle\Sigma, E\rangle$ if A satisfies every equation in E. Then A is called a *model* of $\langle\Sigma, E\rangle$. If $SP = \langle\Sigma, E\rangle$ and $SP' = \langle\Sigma', E'\rangle$ are specifications then SP is a *subspecification* of SP', written $SP \subseteq SP'$, if $\Sigma \subseteq \Sigma'$ and for every model A' of SP', $A'|_{\Sigma}$ is a model of SP.

Formally, we shall define a proof system as any relation between specifications and equations such that the set of equations provable in (i.e., related to) a specification is recursively enumerable. In practice a proof system is a set of inference rules together with a notion of proof leading to such a relation. The recursive enumerability requirement captures the idea that a proof system is an effective procedure for generating the theorems of a specification. It is possible to study proof systems which do not satisfy this requirement (as in infinitary logic [22]) but this topic is outside the scope of this paper.

Definition: A *proof system* is a relation $\vdash \subseteq$ Specifications $\times$ Equations such that if a specification SP is effectively given (i.e., $SP = \langle\Sigma, E\rangle$ where E is recursively enumerable) then the set of provable equations $\{e | SP \vdash e\}$ is recursively enumerable.

Definition: A proof system $\vdash$ is called *complete* for a specification $SP = \langle\Sigma, E\rangle$ if every Σ-equation e which is satisfied in every model of SP (i.e., $SP \models e$) is provable from SP using $\vdash$ (i.e., $SP \vdash e$). Conversely, $\vdash$ is *sound* for SP if every Σ-equa-

$\vdash_{eq} \subseteq$ Specifications × Equations is the smallest relation such that[1]:

1. E is a set of Σ-equations and $t=t' \in E \Rightarrow \langle\Sigma,E\rangle \vdash_{eq} t=t'$
2. (reflexivity) t is a Σ-term $\Rightarrow \langle\Sigma,E\rangle \vdash_{eq} t=t$
3. (symmetry) $\langle\Sigma,E\rangle \vdash_{eq} t=t' \Rightarrow \langle\Sigma,E\rangle \vdash_{eq} t'=t$
4. (transitivity) $\langle\Sigma,E\rangle \vdash_{eq} t=t'$ and $\langle\Sigma,E\rangle \vdash_{eq} t'=t'' \Rightarrow \langle\Sigma,E\rangle \vdash_{eq} t=t''$
5. (substitutivity) $\langle\Sigma,E\rangle \vdash_{eq} t=t'$ and $\langle\Sigma,E\rangle \vdash_{eq} u=u'$ and sort(x)=sort(u)
   $\Rightarrow \langle\Sigma,E\rangle \vdash_{eq} t[u/x]=t'[u'/x]$

[1]This is actually a proof system for ***quantifier-free*** equations. A quantifier-free equation $t = t'$ is implicitly quantified by all the variables which appear in t and t', so it is equivalent to the equation $\forall$ vars(t) $\cup$ vars(t'). $t = t'$.

Fig. 2. The proof system $\vdash_{eq}$.

$\vdash_{GM} \subseteq$ Specifications × Equations is the smallest relation such that:

1. E is a set of Σ-equations and $\forall X.t=t' \in E \Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t'$
2. (reflexivity) X is a set of variables of sorts in Σ and t is a Σ-term with vars(t)$\subseteq$X
   $\Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t$
3. (symmetry) $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t' \Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X.t'=t$
4. (transitivity) $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t'$ and $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t'=t'' \Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t''$
5. (substitutivity) $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t'$ and $\langle\Sigma,E\rangle \vdash_{GM} \forall Y.u=u'$ and $x{:}s \in X$ and sort(u)=s
   $\Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall (X-\{x{:}s\}) \cup Y.t[u/x]=t'[u'/x]$
6. (abstraction) $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t'$ and $x \notin X$ and $s \in$ sorts(Σ) $\Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X \cup \{x{:}s\}.t=t'$
7. (concretion[1]) $\langle\Sigma,E\rangle \vdash_{GM} \forall X.t=t' \Rightarrow \langle\Sigma,E\rangle \vdash_{GM} \forall X-Y.t=t'$
   unless for some sort s there is a Σ-term of sort s with variables in X but no Σ-term of sort s with variables in X-Y.

[1]This version of concretion is due to B. Mahr and J. Loeckx. Regardless, this rule is redundant since it may be derived from rule 5.

Fig. 3. The proof system $\vdash_{GM}$.

tion provable from *SP* using $\vdash$ is satisfied in every model of *SP*.

III. Completeness Results

A. Completeness for Ordinary Equational Specifications

A classical theorem of universal algebra (due to Birkhoff [4]) states that equational logic (the proof system $\vdash_{eq}$ given in Fig. 2) is complete for one-sorted specifications. Goguen and Meseguer [14] have shown that this result extends to the many-sorted case only if equational logic is modified slightly by adding rules to add and delete quantifiers. This modified proof system $\vdash_{GM}$ is given in Fig. 3. They show that the unmodified proof system $\vdash_{eq}$ is not even sound in the many-sorted case. They demonstrate this by means of the following example.

SP = **enrich** Bool **by**
  sorts s
  opns f : s → bool
  eqns $\forall n$: s. f(n) = not(f(n))

(We borrow Clear's **enrich** operation for this example; Bool is given in Fig. 1.) Now, $\vdash_{eq}$ can be used to show that

true = or(f(n),not(f(n)))
  = or(f(n),f(n))
  = f(n)
  = and(f(n),f(n))
  = and(f(n),not(f(n)))
  = false

(recall that $\vdash_{eq}$ is a proof system for quantifier-free equations) and so SP $\vdash_{eq}$ true = false. But this equation does not hold in the model of SP obtained by extending the Σ_{bool}-algebra of Fig. 1 by adding an empty carrier set for sort s. Thus $\vdash_{eq}$ is not sound.

Using $\vdash_{GM}$ the same sequence of deductions shows that SP $\vdash_{GM}$ $\forall n$: s.true = false. This equation *is* satisfied by all models of SP; it is vacuously satisfied when the carrier set for sort s is empty. The concretion rule cannot be applied to remove the quantifiers, since there is no ground term of sort s.

B. Completeness for Initial Models of Specifications

A specification will typically have a number of different (nonisomorphic) models. Some of these will be trivial, and others will contain extra useless values (like the value 2 in the example of Fig. 1). Most approaches to the specification of abstract data types and programs restrict consideration to a special subset of models; probably the best-known example is the "initial algebra" approach of [16] in which a specification is taken to specify only its *initial* models.

Definition: A model *A* of a specification *SP* is an *initial* model of *SP* if for every model *B* of *SP* there is a unique homomorphism $h: A \to B$.

Equivalently, a Σ-algebra *A* is an initial model of $SP = \langle\Sigma, E\rangle$ iff it satisfies the following conditions.

• *"No junk:"* Every element in *A* is the value of some ground Σ-term.

$\|B\|_{\text{bool}} = \{0,1\}$	$\|C\|_{\text{bool}} = \{\Delta,\nabla\}$
$\text{false}_B = 0$	$\text{false}_C = \Delta$
$\text{true}_B = 1$	$\text{true}_C = \nabla$
$\text{not}_B(0) = 1$	$\text{not}_C(\Delta) = \nabla$
$\text{not}_B(1) = 0$	$\text{not}_C(\nabla) = \Delta$
$\text{and}_B(0,0) = \text{and}_B(0,1) = \text{and}_B(1,0) = 0$	$\text{and}_C(\Delta,\Delta) = \text{and}_C(\Delta,\nabla) = \text{and}_C(\nabla,\Delta) = \Delta$
$\text{and}_B(1,1) = 1$	$\text{and}_C(\nabla,\nabla) = \nabla$
$\text{or}_B(0,0) = 0$	$\text{or}_C(\Delta,\Delta) = \Delta$
$\text{or}_B(0,1) = \text{or}_B(1,0) = \text{or}_B(1,1) = 1$	$\text{or}_C(\Delta,\nabla) = \text{or}_C(\nabla,\Delta) = \text{or}_C(\nabla,\nabla) = \nabla$

Fig. 4. Two initial models of Bool = $\langle \Sigma_{\text{bool}}, E_{\text{bool}} \rangle$.

- *"No confusion:"* For every ground Σ-equation e, A satisfies e iff $SP \models e$.

For a proof that these definitions are equivalent and for three other equivalent definitions see [15].

It is well known that the initial models of any specification form an isomorphism class; this allows us to refer to *the* initial model. Category-theoretically speaking, the initial model of a specification SP is the initial object in the category **Mod**(SP) of SP-models and homomorphisms between them. Note that the "no junk" condition corresponds to an induction principle; since all values in the initial model are generated by terms, proof by structural induction on terms is possible. The Σ_{bool} algebra A in Fig. 1 does not satisfy "no junk," so it is not an initial model of Bool. Two initial models of Bool are given in Fig. 4.

When only initial models of specifications are considered, $\vdash_{\text{GM}}$ is complete with respect to ground equations (this is a consequence of the result of [14] mentioned above) but Nourani [25] shows that *no* proof system is sound and complete with respect to nonground equations. (He actually shows that equational logic with induction is not complete, but his proof generalizes easily.) This follows from a consequence of Matijasevič's theorem (see [9]) which states that the set of equations which hold in the standard model of the natural numbers (with 0, +, ×, and −) is not recursively enumerable. It is easy to construct a (one-sorted) specification Nat such that the standard model of the natural numbers is an initial model of Nat (see Fig. 5), so no complete proof system for the initial models of Nat can exist. This result applies to the specification language ACT ONE [10].

C. Completeness for Specifications with Data Constraints

Some languages for algebraic specification adopt what can be seen as a generalization of the initial model approach. They permit specifications to include *data constraints* which must be satisfied (in addition to the equations) by any model of the specification. Each data constraint specifies that for each model, the subalgebra of that model corresponding to a certain subspecification must be a free extension of the subalgebra corresponding to a certain smaller subspecification. The effect is to provide ways to specifying algebras which are initial *relative to* the interpretation of a subspecification, and of specifying algebras which are *extensions* (i.e., do not disturb the carriers but only add new operations) of model of a subspecification. Data constraints make it possible to write *parameterized* specifications like Set-of-X (see Fig. 5) where X may be arbitrary. With data constraints it is also possible to construct *loose* specifications having different nonisomorphic models (see SetChoose in Fig. 5 for an example). This enables a specification to be left purposefully vague so as to allow some freedom to the implementor. The necessary formal definitions below are followed by an intuitive explanation of the meaning of a data constraint.

Definition: Suppose $SP = \langle \Sigma, E \rangle$ and $SP' = \langle \Sigma', E' \rangle$ are specifications with $SP \subseteq SP'$, A and FA are models of SP and SP', respectively, and $\eta_A : A \to FA|_\Sigma$ is a Σ-homomorphism. Then FA is a *free* SP'-model on A with *unit* η_A if for every model B of SP' and Σ-homomorphism $f: A \to B|_\Sigma$ there is a unique Σ'-homomorphism $f^\#: FA \to B$ such that $\eta_A \cdot (f^\#|_\Sigma) = f$.

$$\begin{tikzcd} A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & FA|_\Sigma \arrow[d, "f^\#|_\Sigma"] \\ & B|_\Sigma \end{tikzcd} \qquad \begin{tikzcd} FA \arrow[d, "f^\#"] \\ B \end{tikzcd}$$

SP models $\qquad$ SP' models

According to a result of [23], for all equational specifications SP and SP' and every model A of SP there is a free SP'-model on A. It is easy to prove that all free SP'-models on A are isomorphic; this means we can talk about *the* free SP'-model on A.

Definition: Suppose $SP = \langle \Sigma, E \rangle$ and $SP' = \langle \Sigma', E' \rangle$ are specifications with $SP \subseteq SP'$ and A' is a model of SP'. Let $A = A'|_\Sigma$, let FA be the free SP'-model on A, and let $\epsilon_{A'} = (id_A)^\# : FA \to A'$. Then A' is $(SP \subseteq SP')$-*free* if $\epsilon_{A'}$ is an isomorphism.

This definition is due to [29]. Burstall and Goguen [8] (cf. [21]) give a slightly different definition, saying that A' is $(SP \subseteq SP')$-free if $FA \cong A'$, but there are examples which show that this is not sufficient [6]. Category-theoretically speaking (see [24]) the existence of a free SP'-model FA for every SP-model A determines a functor F: **Mod**(SP) $\to$ **Mod**(SP') which is left adjoint to the forgetful functor $\_|_\Sigma$: **Mod**(SP') $\to$ **Mod**(SP); η and ϵ are the unit and counit, respectively, of the adjunction.

Definition: Given a signature Σ, a Σ-*data constraint dc* is a pair of specifications SP, SP' such that $SP \subseteq SP'$ and sig(SP') $\subseteq \Sigma$ (where sig(SP') is the signature of SP'). A Σ-algebra A *satisfies* a Σ-data constraint $SP \subseteq SP'$ if $A|_{\text{sig}(SP')}$ is $(SP \subseteq SP')$-free.

The above definitions are adapted from [13]. For simplicity we have chosen to adopt a special case which provides all the power necessary for our purposes; these data constraints are essentially the same as the *initial restrictions* of [21]. The incompleteness theorem below also holds for more general notions of data constraint as in [11] and [12] (where they are called *free generating constraints*) and [13].

It is not obvious from the definition what it means intuitively for an algebra to satisfy a data constraint $SP \subseteq SP'$. It turns out that, in analogy with the definition of initial model in Section III-B, there are two conditions which the algebra must satisfy.

- *"No junk:"* Every element of $A|_{\text{sig}(SP')}$ is the value of some sig(SP')-term containing variables only in sorts of SP, for some assignment of values to variables.

- *"No confusion:"* The values of two sig(SP')-terms are the same in A for some assignment of values to variables iff they are forced to be equal by the interpretation of SP in A and the equations of SP'.

```
Bool    =  as in figure 1
dcBool   =  Φ ⊆ Bool
DBool   =  ⟨Bool,{dcBool}⟩

Nat   =  sorts nat
         opns 0 : nat
              succ : nat → nat
              +, ×, - : nat,nat → nat
         eqns ∀n:nat. 0 + n = n
              ∀m,n:nat. succ(n) + m = succ(n + m)
              ∀n:nat. 0 × n = 0
              ∀m,n:nat. succ(n) × m = (n × m) + m
              ∀n:nat. 0 - n = 0
              ∀n:nat. n - 0 = n
              ∀m,n:nat. succ(n) - succ(m) = n - m
dcNat   =  Φ ⊆ Nat
DNat   =  ⟨Nat,{dcNat}⟩

Ident   =  enrich Bool by
             sorts element
             opns eq : element,element → bool
             eqns ∀a:element. eq(a,a) = true
                  ∀a,b:element. eq(a,b) = eq(b,a)
                  ∀a,b,c:element. eq(a,b) and eq(b,c) and not(eq(a,c)) = false
DIdent   =  ⟨Ident,{dcBool}⟩

Set   =  enrich Ident by
           sorts set
           opns φ : set
                add : element,set → set
                ∈ : element,set → bool
           eqns ∀a,b:element, S:set. add(a,add(b,S)) = add(b,add(a,S))
                ∀a:element, S:set. add(a,add(a,S)) = add(a,S)
                ∀a:element. a ∈ φ = false
                ∀a,b:element, S:set. a ∈ add(b,S) = eq(a,b) or a∈S
dcSet   =  Ident ⊆ Set
DSet   =  ⟨Set,{dcBool,dcSet}⟩

SetChoose   =  enrich Set by
                 opns  choose : set → element
                 eqns  ∀a:element, S:set. choose(add(a,S)) ∈ add(a,S) = true
DSetChoose   =  ⟨SetChoose,{dcBool,dcSet}⟩
```

Fig. 5. Data specifications.

Again, the "no junk" condition corresponds to an induction principle. The idea of "no confusion" is the same as in the case of initial algebras (as few equations as possible are satisfied) although the condition itself is more complicated. Some examples of data constraints and algebras which satisfy them are given in Fig. 5.

Definition: A *data specification* is a specification $\langle\Sigma, E\rangle$ together with a set of Σ-data constraints *DC*. A Σ-algebra *satisfies* a data specification $\langle SP, DC\rangle$ if it satisfies *SP* and every data constraint in *DC*.

Referring to Fig. 5, the models of DBool are just the initial models of Bool as in Fig. 4. In general, if *SP* is a specification then the models of the data specification $\langle SP, \{\Phi \subseteq SP\}\rangle$ will be the initial models of *SP*. The models of DNat are then the initial models of Nat and hence are isomorphic to the standard model of the natural numbers. Every model of DIdent will consist of a model of DBool extended by some arbitrary carrier for the sort *element* together with an equivalence relation *eq*. Note that the initial models of Ident are rather uninteresting since they all have an empty carrier for the sort *element*. The constraint dcSet says that in each model A of DSet, $|A|_{\text{set}} \cong \mathcal{F}(|A|_{\text{ident}})$ (the set of finite subsets of $|A|_{\text{ident}}$) with the appropriate interpretations for the operators ϕ, *add*, and $\in$. The models of DSetChoose are the same as the models of DSet except that each model includes an operaton which will select an arbitrary element from a nonempty set. DSetChoose is a loose specification because the result of *choose* is only partially specified; the choice of which element of a set to return (and the result when the argument is ϕ) is left up to the implementor. Specification languages like Clear provide a convenient syntax for presenting data specifications so users never have to write data constraints explicitly. We have borrowed Clear's **enrich** operation for the examples in Fig. 5 but we have chosen to display the data constraints.

For data specifications $\vdash_{\text{GM}}$ is not complete with respect to ground equations, even when induction is permitted. This fact is demonstrated by the following simple example:

DSP = **enrich** DNat **by**
opns f : nat → nat
eqns $\forall n$: nat. $\mathrm{f}(n) = 2 \times \mathrm{f}(n+1)$

where 1 and 2 are abbreviations for *succ*(0) and *succ*(*succ*(0)). For all models A of DSP we have $A \models \forall\phi.\mathrm{f}(0) = 0$ (recall that the function associated with *f* must be total and the sort *nat* does not include an "infinite" element). But this equation is not provable using $\vdash_{\text{GM}}$ with induction; this may be shown by induction and case analysis on the terms which may be derived from f(0).

Note that DSP is *not* a counterexample for the completeness of $\vdash_{\text{GM}}$ with respect to ground equations for initial models of equational specifications (Section III-B). The initial model of DSP viewed as a purely equational specification (i.e., ignoring the data constraint dcNat) does not satisfy $\forall\phi.\mathrm{f}(0) = 0$. It is the "no junk" requirement (with respect to the operators of DNat, not DSP!) imposed by the data constraint dcNat which makes the difference by forcing the value of f(0) to be equal to the value of some term of the form $\mathrm{succ}^j(0)$ (for $j \geqslant 0$).

It is possible to prove the equation $\forall\phi.\mathrm{f}(0) = 0$ in DSP if a strong enough proof system (including proof by contradiction) is used. But for some data specifications there is no proof system which is strong enough to prove all true ground equations.

Theorem: There exists no proof system for data specifications which is sound and complete with respect to ground equations.

Proof: Proposition 4 of [3] states that for any total recursive function $f: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ there is a finite data specification SP_f having as its only model (to within isomorphism) an algebra A_f consisting of the natural numbers $\mathbb{N}$ enriched by the function

$$ex_f(y) = \begin{cases} 1 & \text{if } \exists x \in \mathbb{N} \text{ such that } f(x,y) > 0 \\ 0 & \text{otherwise.} \end{cases}$$

Suppose *f* is the total recursive function

$$f(x,y) = \begin{cases} 1 & \text{if } x \text{ codes a convergent computation of } \varphi_y(y) \\ 0 & \text{otherwise} \end{cases}$$

(where φ_y is the partial recursive function with Gödel number y). Then ex_f is the characteristic function of the complete recursively enumerable set K (see [26]) so ex_f is not recursive and therefore its graph is not recursively enumerable. Hence the set of equations $\forall\phi.ex_f(n) = m$ true in A_f is not r.e., where n, m are ground terms ($\mathrm{succ}^j(0)$ for some j). Since for any proof system $\vdash$ the set of theorems which can be derived from a specification is r.e., there must be ground terms n, m such that $A_f \models \forall\phi.ex_f(n) = m$ (so $SP_f \models \forall\phi.ex_f(n) = m$ since A_f is the only model of SP_f) but $SP_f \not\vdash \forall\phi.ex_f(n) = m$. □

The key to this proof is the result of [3] which shows that it is possible to encode existential (and universal) quantifiers over the natural numbers in a data specification. This incompleteness theorem applies to the specification language described in [21] and the specification languages Clear [8] and LOOK [31], as well as to specification languages adopting the free generating constraints of [11] and [12].

D. Completeness for Specifications with Hierarchy Constraints

Certain specification languages employ a weaker version of data constraints, sometimes called *hierarchy constraints.* While a data constraint specifies that a certain subalgebra A' of a model must be a free extension of a certain smaller subalgebra A—that is, that the model must satisfy the "no junk" and "no confusion" conditions given in Section III-C—a hierarchy constraint requires only that the model satisfies the "no junk" condition which says that it must be possible to *generate* all values in A' from the values in A using the operations in the signature of A'. This in itself is not enough to eliminate all undesirable models since, e.g., degenerate models in which each carrier is empty or contains only a single element (depending on the signature) will satisfy any set of equations and hierarchy constraints. Therefore, specification languages which employ hierarchy constraints either allow specifications to contain *inequations* (see [1], [28]) or require that all specifications include the subspecification Bool (see Fig. 1) and that all models satisfy the inequation $\forall\phi.\text{true} \neq \text{false}$ (see [27]). We will adopt the latter approach. Like data constraints, hierarchy constraints make it possible to construct parameterized specifications and loose specifications. An advantage of hierarchy constraints over data constraints is that they make it much easier to write certain kinds of loose specifications; this point is discussed in a comment near the end of this section. On the other hand, it is sometimes necessary to include some extra operators and equations in a specification with hierarchy constraints in order to eliminate trivial models.

Technically, the difference between data constraints and hierarchy constraints is solely in the way that satisfaction of a constraint by an algebra is defined. Compare the following definition of an $(SP \subseteq SP')$-hierarchical algebra (adapted from [13]) with the definition of an $(SP \subseteq SP')$-free algebra in Section III-C.

Definition: Suppose $SP = \langle \Sigma, E \rangle$ and $SP' = \langle \Sigma', E' \rangle$ are specifications with $SP \subseteq SP'$ and A' is a model of SP'. Let $A = A'|_\Sigma$, let FA be the free SP'-model on A, and let $\epsilon_{A'} = (id_A)^{\#}: FA \to A'$. Then A' is $(SP \subseteq SP')$*-hierarchical* if $\epsilon_{A'}$ is a surjection.

Definition: Given a signature Σ, a Σ*-hierarchy constraint hc* is a pair of specifications SP, SP' such that $SP \subseteq SP'$ and $\text{sig}(SP') \subseteq \Sigma$. A Σ-algebra A *satisfies* a Σ-hierarchy constraint $SP \subseteq SP'$ if $A|_{\text{sig}(SP')}$ is $(SP \subseteq SP')$-hierarchical.

Definition: A *hierarchical specification* is a specification $\langle \Sigma, E \rangle$ together with a set of Σ-hierarchy constraints HC, where $\text{Bool} \subseteq \langle \Sigma, E \rangle$ and $(\Phi \subseteq \text{Bool}) \in HC$. A Σ-algebra A *satisfies* a hierarchical specification $\langle SP, HC \rangle$ if it satisfies SP and every hierarchy constraint in HC, and $\text{true}_A \neq \text{false}_A$.

The models of the hierarchical specifications in Fig. 6 are exactly the same as the models of the corresponding data specifications in Fig. 5 (except HNat models have an extra operation, of course). HNat needs $\leqslant$ and the equations which define it in order to eliminate trivial models [otherwise some models will satisfy, e.g., $\forall\phi.0 = \text{succ}(0)$]. The same result could have been accomplished by adding instead $\geqslant$ or an equality predicate. HSet needs no extra operations to eliminate trivial models (satisfying, e.g., $\forall n: \text{element}.\phi = \text{add}(n, \phi)$); $\in$ is enough to induce the inequations necessary to make all models isomorphic to the standard model.

```
Bool  =  as in figure 1
hcBool  =  Φ ⊆ Bool
HBool  =  ⟨Bool,{hcBool}⟩

Nat'  =  enrich Bool by
           sorts nat
           opns 0 : nat
                succ : nat → nat
                +, ×, - : nat,nat → nat
                ≤ : nat,nat → bool
           eqns ∀n:nat. 0 + n = n
                ∀m,n:nat. succ(n) + m = succ(n + m)
                ∀n:nat. 0 × n = 0
                ∀m,n:nat. succ(n) × m = (n × m) + m
                ∀n:nat. 0 - n = 0
                ∀n:nat. n - 0 = n
                ∀m,n:nat. succ(n) - succ(m) = n - m
                ∀n:nat. 0 ≤ n = true
                ∀n:nat. succ(n) ≤ 0 = false
                ∀m,n:nat. succ(n) ≤ succ(m) = n ≤ m
hcNat  =  Φ ⊆ Nat'
HNat  =  ⟨Nat',{hcBool,hcNat}⟩

Ident  =  as in figure 5
HIdent  =  ⟨Ident,{hcBool}⟩

Set  =  as in figure 5
hcSet  =  Ident ⊆ Set
HSet  =  ⟨Set,{hcBool,hcSet}⟩

SetChoose  =  as in figure 5
HSetChoose  =  ⟨SetChoose,{hcBool,hcSet}⟩
```

Fig. 6. Hierarchical specifications.

It is interesting to note what would have happened if we had omitted the first two equations of Set (i.e., $\forall a, b$: element, S: set.add(a, add(b, S)) = add(b, add(a, S)) and $\forall a$: element, S: set.add(a, add(a, S)) = add(a, S)). All the models of DSet would then be isomorphic to the standard model of lists. But the models of HSet would not form an isomorphism class. They would include models which satisfy both equations (sets), models satisfying neither equation (lists), models satisfying only the first equation (bags), models which satisfy only the second equation (a strange hybrid of sets and lists) and also models which satisfy either or both of the equations only for some values. Note that all these models satisfy the equations defining $\in$, and trivial models are excluded. It is possible to construct a data specification having this class of models (with some auxiliary sorts and operators) but the construction is rather unnatural.

The proof of the incompleteness theorem of Section III-C remains valid for specification languages which use hierarchy constraints rather than data constraints since the crucial result from [3] used in the proof holds in this case as well. Hence, the theorem holds for the specification languages CIP-L [1], hierarchical Clear [27] and ASL [28]. It also holds for more general notions of hierarchy constraints, such as the *generating constraints* of [11] and [12].

IV. Conclusion

Table I summarizes the results we have discussed.

Although the discussion has been confined to equational specifications, it is clear that all the incompleteness results presented also apply in the context of logical systems such as first-order logic with equality and Horn-clause logic in which equations are expressible. They also hold when algebras are allowed to contain partial functions as in [7], provided the logic includes definedness constraints in some form. The particular form of data/hierarchy constraints and the restriction to initial models are also not of essential importance to these results; incompleteness with respect to all equations depends only on the ability to restrict in some way to models satisfying "no junk" (needed, e.g., to specify the standard model of the

TABLE I

	Complete Proof System with Respect to Ground Equations	Complete Proof System with Respect to All Equations
One-Sorted Specifications	$\vdash_{eq}$	$\vdash_{eq}$
Many-Sorted Specifications	$\vdash_{GM}$	$\vdash_{GM}$
Initial Models of Specifications	$\vdash_{GM}$	None Exists
Data Specifications	None Exists	None Exists
Hierarchical Specifications	None Exists	None Exists

natural numbers), while incompleteness with respect to ground equations requires the ability to restrict to models having a given subalgebra satisfying "no junk" (needed for building specifications in a hierarchical fashion).

The incompleteness results presented here are of practical as well as theoretical interest. A central issue in any methodology for program development from specifications is the problem of proving the correctness of programs with respect to their specifications, which may sometimes be decomposed into proving the correctness of a number of refinement steps. This problem reduces to that of performing proofs in specifications. Another use of theorem proving in program development is in analyzing and understanding specifications, as suggested in [19]. A different application is the incestuous one of checking the semantic well-formedness of specifications, for example making sure that parameterized specifications are only applied to appropriate arguments (the analog of type checking at this level requires a theorem proving capability).

The incompleteness results say that under any reasonable specification formalism, an equation may be satisfied in all models of a specification even though it cannot be proved to follow from the specification. Incompleteness with respect to ground equations has more serious consequences. For example, in testing whether a program satisfies its specification by checking the outputs obtained for particular input values, a given output may be correct although there may be no way of determining from the specification that it is correct. However, these incompleteness results do not demonstrate the infeasibility of using specifications in program development. They only indicate certain limitations which must be taken into account, just as the halting problem places a limit on our ability to formally reason about programs.

Acknowledgment

The authors thank F. Nourani, D. Rydeheard, A. Tarlecki, and (especially) M. Wirsing for helpful discussions, and T. Maibaum for useful comments on a draft version.

References

[1] F. L. Bauer *et al.* (the CIP Language Group), "Report on a wide spectrum language for program specification and development," Technische Univ., München, West Germany, Rep. TUM-I8104, 1981.

[2] J. Benabou, "Structures algebriques dans les categories," *Cahiers de Topologie et Geometrie Differentiel*, vol. 10, pp. 1-24, 1968.

[3] J. A. Bergstra, M. Broy, J. V. Tucker, and M. Wirsing, "On the power of algebraic specifications," in *Proc. 10th Int. Symp. Math. Foundations of Comput. Sci.*, Strbske Pleso, Czechoslovakia. New York: Springer LNCS 118, pp. 193-204, 1981.

[4] G. Birkhoff, "On the structure of abstract algebras," *Proc. Cambridge Phil. Soc.*, vol. 31, pp. 433-454, 1935.

[5] G. Birkhoff and D. Lipson, "Heterogeneous algebras," *J. Combinatorial Theory*, vol. 8, no. 1, pp. 115-133, 1970.

[6] S. L. Bloom and E. G. Wagner, "Many-sorted theories and their algebras with some applications to data types" (draft), presented at the US-French Joint Symp. Application of Algebra to Language definition and Compilation, Fontainebleau, France, 1982; proceedings to be published.

[7] M. Broy and M. Wirsing, "Partial abstract types," *Acta Inform.*, vol. 18, pp. 47-64, 1982.

[8] R. M. Burstall and J. A. Goguen, "The semantics of Clear, a specification language," in *Proc. Advanced Course Abstract Software Specifications*, Copenhagen, Denmark. New York: Springer LNCS 86, pp. 292-332, 1980.

[9] M. Davis, Y. Matijasevič, and J. Robinson, "Hilbert's tenth problem. Diophantine equations: Positive aspects of a negative solution," in *Proc. Symp. Pure Math.*, vol. 28, 1976.

[10] H. Ehrig, W. Fey, and H. Hansen, "ACT ONE: An algebraic specification language with two levels of semantics," Institut für Software und Theoretische Informatik, Technische Univ., Berlin, West Germany, Rep. 83-03, 1983.

[11] H. Ehrig, E. G. Wagner, and J. W. Thatcher, "Algebraic constraints for specifications and canonical form results" (draft version), Institut für Software und Theoretische Informatik, Technische Univ., Berlin, West Germany, Rep. 82-09, 1982.

[12] —, "Algebraic specifications with generating constraints," in *Proc. 10th Int. Colloq. Automata, Lang. Programming*, Barcelona, Spain. New York: Springer LNCS 154, pp. 188-202, 1983.

[13] J. A. Goguen and R. M. Burstall, "Introducing institutions," in *Proc. Logics of Programming Workshop*, E. Clarke, Ed., Carnegie-Mellon Univ., Pittsburgh, PA, 1983.

[14] J. A. Goguen and J. Meseguer, "Completeness of many-sorted equational logic," *Sigplan Notices*, vol. 16, no. 7, pp. 24-32, 1981; see extended version in vol. 17, no. 1, pp. 9-17; SRI Int., Tech. Rep. CSL-135, May 1982.

[15] —, "An initiality primer," SRI Int., Draft Rep., 1983.

[16] J. A. Goguen, J. W. Thatcher, and E. G. Wagner, "An initial algebra approach to the specification, correctness, and implementation of abstract data types," IBM Res. Rep. RC 6487, 1976; see also *Current Trends in Programming Methodology, Vol. 4: Data Structuring*, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1978, pp. 80-149.

[17] G. Grätzer, *Universal Algebra*, 2nd ed. New York: Springer, 1979.

[18] J. V. Guttag, "The specification and application to programming of abstract data types," Ph.D. dissertation, Univ. Toronto, Toronto, Ont., Canada, 1975.

[19] J. V. Guttag and J. J. Horning, "Formal specification as a design tool," in *Proc. 7th ACM Symp. Principles of Programming Lang.*, Las Vegas, NV, 1980.

[20] P. J. Higgins, "Algebras with a scheme of operators," *Mathematische Nachrichten*, vol. 27, pp. 115-132.

[21] U. L. Hupbach, H. Kaphengst, and H. Reichel, "Initial algebraic specification of data types, parameterized data types, and algorithms," VEB Robotron, Zentrum für Forschung und Technik, Dresden, East Germany, 1980.

[22] C. R. Karp, *Languages with Expressions of Infinite Length.* Amsterdam, The Netherlands: North-Holland, 1964.

[23] F. W. Lawvere, "Functorial semantics of algebraic theories," in *Proc. Nat. Acad. Sci.*, vol. 50, pp. 869-872, 1963.

[24] S. MacLane, *Categories for the Working Mathematician.* New York: Springer, 1971.

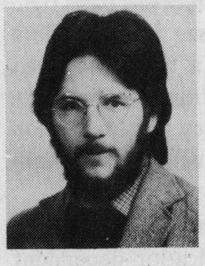
[25] F. Nourani, "On induction for programming logic: Syntax, semantics, and inductive closure," *Bull. EATCS*, vol. 13, pp. 51-64, 1981.

[26] H. Rogers, *Theory of Recursive Functions and Effective Computability.* New York: McGraw-Hill, 1967.

[27] D. T. Sannella and M. Wirsing, "Implementation of parameterised specifications," Dep. Comput. Sci., Univ. Edinburgh, Scotland, Rep. CSR-103-82; see extended abstract in *Proc. 9th Int. Colloq. Automata, Lang. Programming*, Aarhus, Denmark. New York: Springer LNCS 140, pp. 473-488, 1982.

[28] —, "A kernel language for algebraic specification and implementation," Dep. Comput. Sci., Univ. Edinburgh, Scotland, Rep. CSR-131-83; see extended abstract in *Proc. Int. Conf. Foundations of Computation Theory*, Borgholm, Sweden. Springer LNCS 158, p. 413–427, 1983.
[29] J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Data type specification: Parameterization and the power of specification techniques," presented at the SIGACT 10th Symp. Theory of Comput., San Diego, CA, 1978; see also *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 4, pp. 711–732, 1982.
[30] S. N. Zilles, "Algebraic specification of data types," Computation Structures Group, Lab. Comput. Sci., Massachusetts Inst. Technol., Cambridge, Memo 119, 1974.
[31] S. N. Zilles, P. Lucas, and J. W. Thatcher, "A look at algebraic specifications," IBM Res. Rep. RJ 3568, 1982.

David B. MacQueen, photograph and biography not available at the time of publication.



Donald T. Sannella received the B.S. degree from Yale University, New Haven, CT, in 1977, the M.S. degree from the University of California at Berkeley in 1978, and the Ph.D. degree from the University of Edinburgh, Edinburgh, Scotland, in 1982, all in computer science.

He is currently a Postdoctoral Research Fellow at the University of Edinburgh. His research interests include algebraic program specification, functional programming languages, and mechanical theorem proving.

Atomic Actions and Resource Coordination Problems Having Nonunique Solutions

MUKUL K. SINHA

Abstract—The concept of atomic actions is decomposed into database-dependent atomic actions and application-dependent atomic actions. There is a broad class of application-dependent atomic actions that can have nonunique solutions. These are resource coordination problems and are classified as problems of *NU* class. It is argued that a transaction modeling a problem of *NU* class provides lower concurrency. A concept of coordination is proposed which can model a broad range of *NU* class problems. An object model and a protocol are suggested which utilize the nonunique character of the solution to provide higher concurrency.

Index Terms—Access synchronization, algorithms, atomic actions, concurrency control, crash recovery, design, performance.

I. Introduction

An *atomic action* is composed of a set of primitive actions on different data items which cannot be decomposed from the point of view of computation outside of the atomic action [8], i.e., the intermediate states of data items must not be visible outside the computation of the atomic action. An atomic action remains atomic in face of failure. Thus, an atomic action has to be a unit of *concurrency control* [1] as well as a unit of *recovery* [9].

Manuscript received May 27, 1983.

The author is with the National Centre for Software Development and Computing Techniques, Tata Institute of Fundamental Research, Bombay 400 005, India.

In a database system, users access shared data under the assumption that the database is in *consistent* state, i.e., the values of data items satisfy a set of assertions, the *consistency constraints* of the database. A user of a database may need to temporarily violate the consistency of the database, but at the end of a set of actions he must restore the database to a consistent state. For this reason, actions are grouped together to form a *transaction* [4]. Transactions are units of consistency and the consistency constraints must apply at the start and the end of transaction processing, not necessarily during the transaction processing. Thus, in general, intermediate states of the database must not be visible outside the transaction, i.e., a transaction is an atomic action as well. Since a transaction satisfies the consistency constraints of a database, it is a *database-dependent atomic action.*

In literature, whenever techniques for synchronization and recovery in decentralized computer system are discussed [7], [10], they always refer to one type of atomic actions, that is the transaction. There is a broad class of problems in a distributed system environment which requires atomicity at application level. Consider an atomic action composed of two primitive actions {*update X*; *update Y*} which updates data items *X* and *Y*. If updating the data item *X* independently (or updating the data item *Y* independently) does not violate any integrity/consistency constraint of the database, the database is never inconsistent during the execution of the atomic action.