Semantics and Correctness Proofs for Programs with Partial Functions

Alexander Yakhnis

and

Vladimir Yakhnis

To be submitted at
ACM SIGSOFT'96, Fourth Symposium on the Foundations of Software Engineering
San Francisco, California, 16-18 October 1996

This work was supported by the United States Department of Energy under Contract DE-AC04-94AL85000.

DISCLAIMER

Semantics and Correctness Proofs for Programs with Partial Functions

Alexander Yakhnis
Command and Control Software Department
Sandia National Laboratories, MS-0535
1515 Eubank SE, Albuquerque, NM 87123
Tel: (505) 844-0277 E-mail: aryakhn@sandia.gov

Vladimir Yakhnis
Command and Control Software Department
Sandia National Laboratories, MS-0535
1515 Eubank SE, Albuquerque, NM 87123
Tel: (505) 844-8672 E-mail: vryakhn@sandia.gov

ABSTRACT This paper represents a portion of our work on specification, design, and implementation of safety-critical systems. Obviously, it is very desirable to have rigorous proofs of functional, safety, and security properties for nuclear and chemical reactors, defense, medical, and communication systems. A natural approach to this problem, once all the requirements are captured, would be to state the requirements formally and then either to prove (preferably via automated tools) that the system conforms to spec (program verification), or to try to simultaneously generate the system and a mathematical proof that the requirements are being met (program derivation). An obstacle to this is frequent presence of partially defined operations within the software and its specifications. Indeed, the usual proofs via first order logic presuppose everywhere defined operations. Recognizing this problem, David Gries, in "The Science of Programming", 1981, introduced the concept of partial functions into the mainstream of program correctness and gave hints how his treatment of partial functions could be formalized. Still, however, existing theorem provers and software verifiers have difficulties in checking software with partial functions, because of absence of uniform first order treatment of partial functions within classical 2-valued logic.

Several rigorous mechanisms that took partiality into account were introduced [Wirsing 1990, Breu 1991, VDM 1986, 1990, etc.]. However, they either did not discuss correctness proofs or departed from first order logic. To fill this gap, we provide a semantics for software correctness proofs with partial functions within classical 2-valued 1st order logic. We formalize the Gries treatment of partial functions and also cover computations of functions whose argument lists may be only partially available. An example is nuclear reactor control relying on sensors which may fail to deliver sense data. Our approach is sufficiently general to cover correctness proofs in various implementation languages (C/C++, Eiffel, etc.)

KEYWORDS: correctness proofs, partial operations, 1st order logic, Hoare triple, Dijkstra language.

1. INTRODUCTION

1.1. The Goals of the Paper

1.1.1. Extending Logical Connectors

Consider a typical safety-critical system: a nuclear reactor control system. One of its safety subsystems must issue a shutdown command once the sensors detect that neutron density is above certain critical value *crit*. For simplicity, suppose there are 2 sensors whose measurements are m and n. If the Boolean value of F(m,n)=(n≥*crit* OR m≥*crit*) is true, the shutdown must follow. If any of the sensors fail to deliver a value, the corresponding inequalities do not make mathematical sense, and therefore the logical value of the Boolean expression is undefined in the classical 2-valued (**true**, **false**) logic. This means that we have to extend the meaning of the logical connector OR to the case when one of its Boolean inputs is undefined. Logical connectors such as OR can be extended over 3-valued Boolean domain in a variety of ways. E.g., [Gries 1981] and [Jones 1990] introduced a 3rd Boolean value "undefined", however, while Gries provided an asymmetric extension of OR, Jones provided a symmetrical one. The above example corresponds to the "symmetric" extension of OR. Since a single safety-critical system may need several different extensions of each classical Boolean connector, we need to provide a uniform treatment of all such extensions. This is one of the goals of this paper.

1.1.2. Formalizing the D. Gries Technique for Correctness Proofs

Consider the following example from [Gries 1981]. What is the logical value of (x = 0 OR y/x = 5) when x = 0?. The first disjunct holds, while the second has no standard meaning since y/0 is undefined. If, however, we choose an arbitrary value "w" for y/0 thus extending division to a total function, the

logical value of (x = 0 OR y/x = 5) will be **true** independently of the choice of w. This is the essence of the following idea of Gries:

- If all partial functions in a formula are somehow extended to total functions, then we can try to prove the formula as if the functions were indeed total;
- If during the proof we would never take an advantage of the values extending the partial functions into total, the proof would be valid.

This technique is very convenient since it enables the classical 2-valued first order logic to be applied formulas with partial functions. However, in order to make this technique both rigorous and amenable to automation, the following questions must be answered:

- which formulas may be treated in this fashion?
- since the classical Tarski's semantics of classical first-order logic formulas does not treat partial functions, in which sense can we speak about the validity of the proofs within the Gries technique?
- is there a rigorous meta-proof of the validity of the technique?

Another goal of the paper is to provide positive answers to the above questions.

1.1.3. Functions whose Argument Lists May Be only Partially Available

Curiously enough, the mere usage of partial functions introduces another problem with semantics of proofs within the realm of total functions. Consider the "selection" function (b ? x : y) from C/C++. It is obviously a total function. However, what is the meaning of (**true** ? 1 : 1/0)? It is 1, even if the third argument is undefined. Thus, although, by itself, (b ? x : y) is total, its usage does not conform to the classical Tarski's semantics, since it does not allow undefined arguments. The third goal of the paper is to extend the classical 2-valued 1st order logic to such usages of total functions.

1.1.4. Meaning and Correctness Proofs of Hoare Triples with Partial Functions

In order to allow for partial functions within Hoare triples of the form $\{P\}\mathscr{R}\{Q\}$, we extend their "total correctness" meaning as follows:

- the assertion $\{P\}\mathscr{R}\{Q\}$ is
  - **true**, if precondition P is both *defined* and **true**, then the program $\mathscr{R}$ terminates and upon its completion the postcondition Q is both *defined* and **true**.
  - **false**, otherwise.

Although the Hoare triples are the major mechanism for proving correctness of terminating programs, the current state of the correctness proofs practice does not adequately address Hoare triples with partial functions. Consider an example from [Kaldewaij 1990], an excellent book on program correctness and derivation. It is suggested there (and in many other books and papers, e.g., [Gries 1980; Cohen 1990; Yakhnis, Farrell, Shultz 1994], etc.) that in order to prove a Hoare triple of the form P{x:=E}Q, one has to show that $P \Rightarrow Def.E \wedge Q(x/E)$, where Q(x/E) is the result of substitution of E for x, holds. There are three problems with such treatment:

- the expression transformer Def is not formally defined. This makes the approach less amenable to automation;
- the meaning of connector $\wedge$ must be extended to cover undefined inputs. This is done in a limited form in several works (e.g., [Gries 1980; Yakhnis, Farrell, Shultz 1994]);
- even if $\wedge$ is extended, the formula would become undefined if P or Q contain partial functions. Moreover, the occurrence of partial functions in P or Q is quite common.

For instance, consider a program using a one dimensional array f of length 100. Then f is a partial function (over integers) whose domain is the segment [1..100]. If we would want to require some property of f upon the completion of the program then f must be included in the postcondition Q. E.g., {**true**}n := 101{f(n)>0} is **false**, since upon the execution of n := 101 the postcondition f(n)>0 is not defined. Kaldewaij's formula gives us $\textbf{true} \Rightarrow Def.(101) \wedge f(102)>0$, which is equivalent to f(101)>0

which is neither **true** nor **false**. Thus an automatic checking based on this proof rule would not yield a definite answer.

Our final goal is to provide a formal definition of Def and to modify the Hoare and Dijkstra proof rules for all the program connectors, so that one would be able to find by automatic means the logical values of the Hoare triples in the presence of partial functions.

1.2. The Existing Research on Partial Functions

Many researchers worked in the area of partial functions and their applications in computing [Stephen Kleene 1936-1950, David Gries 1981, 1983, Horst Reichel 1987, Cliff Jones (VDM) 1986, 1990, Martin Wirsing 1990, Ruth Breu 1991, Yuri Gurevich 1992].

In order to reason about partial functions, 3-valued logic was used by Kleene in his classical "Introduction to Mathematical Logic", 1952. Kleene described several 3-valued logics developed by him (1938) and others (e.g., the Lukacevich logic 1920). One of this 3-valued logics is identical to that of Jones 1986, 1990, however, Kleene's purpose was to elucidate partial functions in recursion theory, rather than to reason about software. Beginning from Gries all the authors used 3-valued logics described in [Kleene 1952] and introduced various versions of explicit domains for partial functions.

Our explicit domains are substantially different from the ones previously considered:

- to accommodate computations with functions whose argument lists may be only partially available, we impose an additional structure on the explicit domains;
- we represent the explicit domains for atomic functions in a form suitable for uniform automated computation of domains for compound terms built up from the atomic partial functions;
- we proved that all 3-valued logics of atomic Boolean functions [described in Kleene 1952] can be reformulated using classical 2-valued 1st order logic;
- we provide a uniform reduction of Hoare triples with partial functions to formulas of classical 2-valued 1st order logic making them more amenable to automated proofs.

1.3. Outline of Our Selected Results

As we have demonstrated in the previous examples, while overall computation may be correct, some of the subordinate computations may not yield any definite result or even may not terminate. We would like to make definite conclusions about correctness of such software in a uniform way within 2-valued 1st order logic. This would be the basis for automated verification of correctness of software. We model computations that may not yield any definite result or may not terminate by means of partial functions. Partial functions were dealt with in mathematics rigorously for quite a long time. However, with respect to software there is more difficulty in handling them. This is because, while in mathematics overstepping the domain of a partial function is prohibited and is watched over very closely, computation of a partial function outside its domain on computers is a common occurrence. Another common occurrence is a computation of a "total function" on an invalid input which is the same as regarding the function as partial on a larger domain. This makes it a challenge to reason in an uniform and practical way about using partial functions in software engineering.

The simplified outline of our approach is as follows. For every pair consisting of a piece of software and a requirement imposed upon it (either of which may contain partial functions), we construct a classical 2-valued 1st order logic formula such that:

- all the symbols denoting partial functions are considered as if they denote total functions. Each total denotation coincides with the corresponding partial one over the domain of the partial denotation;
- if it has classical 1st order logic proof then the software is correct with respect to the requirement;
- if its negation has such proof, then the software is faulty with respect to the requirement;
- if neither of the 2 proofs above exist, then nothing can be said about the software;
- Now, in order to check automatically such software, we need to run an automatic theorem prover on the formula.

We proceed as follows. We extend the universe over which we consider the functions occurring in a piece of software by a single value denoting undefined value $\perp$. To every partial atomic function, say, f(x, y) we attach another atomic function Edom.f(z, x, w, y) which is boolean-valued and total over the

extended universe and represents a classical formula of 1st order 2-valued logic. We do not distinguish later between that formula and the function. Here z, and w are Boolean variables (i.e., they are taken from $\mathbb{B}$ = {**true**, **false**}) and "Edom" stands for "explicit domain". The meaning of Edom.f(z, x, w, y) is the following:

- the standard set-theoretical domain may be computed as Dom.f = {(x, y) | Edom.f(**true**, x, **true**, y) = **true**};
- if Edom.f(**true**, x_0, **false**, y) = **true** then $(x_0, y) \in$ Dom.f, $f(x_0, y)$ does not depend on y and, moreover, $f(x_0, y)$ may be computed without knowing y;
- if Edom.f(**false**, x, **true**, y_0) = **true** then $(x, y_0) \in$ Dom.f, $f(x, y_0)$ does not depend on x and, moreover, $f(x, y_0)$ may be computed without knowing x.

For every compound term $t = f(t_1, ..., t_k)$ (where $t_1, ..., t_k$ are other terms) we inductively define the expression transformer Def by Def.t = Edom.f(Def.t_1, t_1,..., Def.t_k, t_k). It follows by induction that Def.t is a total Boolean valued function over the extended universe representing a formula of classical 2-valued 1st order logic.

Theorem 1. If Def.t = **true** then the value of t may be computed without attempts to find the values of atomic partial functions outside of their domains.

□

Theorem 2. Let φ be a formula of classical 2-valued 1st order logic (φ does not have free variables. Note that formulas representing Hoare triples are of this kind.) Suppose that some of the functional symbols of φ are interpreted as partial functions over the original universe. Then the following holds:

- If a classical 2-valued 1st order logic proof of Def.φ (¬Def.φ) exists, then Def.φ (¬Def.φ) is **true** over the original universe.

□

Remark 1. Without Theorem 2 the existence of the classical proof mentioned above implies only that the formula Def.φ (¬Def.φ) is **true** over the extended universe.

□

We say that φ is total if Def.φ is **true**. Otherwise, we say that φ is not defined.

Theorem 3. Let φ be as in Theorem 2. If φ is total then:

if a classical 2-valued 1st order logic proof of φ (¬φ) exists, then φ (¬φ) is **true** over the original universe.

□

Theorem 4. A Hoare triple $\{P\}\mathscr{R}\{Q\}$ holds if there is a well-formed classical 2-valued 1st order logic proof of Def(P) ∧P ⇒ wpp($\mathscr{R}$, Q) using our proof rules. Here, wpp($\mathscr{R}$, Q) is the "weakest precondition for $\mathscr{R}$, Q in the presence of partial functions". The table rules defining wpp is at the end of the paper.

□

Thus in order to check a piece of software $\mathscr{R}$ with respect to a precondition P and postcondition Q, it is sufficient to run a theorem prover on the formula from Theorem 4 for the corresponding Hoare triple. If there is a classical proof of the formula, the software is correct. If there is a classical proof of the negation of the formula, the software is faulty. Otherwise, it is inconclusive.

1.4. Sorted Partial Algebras

In our approach we rely on the notion of sorted partial algebras or equivalent notions, as described in several works (Breu, R. 1991; Wirsing, M. 1990; Gurevich 1992, etc.). We will provide a brief introduction to these notions.

According to a standard definition, a (mathematical) structure (sometimes also called "mathematical model") is a combination of a set (called the universe) and a collection of functional and relational symbols (called the signature), where each element of the signature is associated with an n-ary function or relation (for various n's) defined in terms of the universe. In addition, it is assumed that each function of arity n is defined everywhere on $\mathbb{U}\times...\times\mathbb{U}$ (n times) (also denoted as $\mathbb{U}^n$), where $\mathbb{U}$ is the universe (i.e., each functions is total). The signature, the logical symbols (i.e., ¬, ∧, ∨, ⇒, etc.), and a collection of

variables, form the first order language associated with the structure. First order structures disallow quantifications ($\forall$ and $\exists$) over functions and relations.

In order to somewhat relax the prohibitions on partial functions and quantifications over functions and relations the notion of sorted structures was developed. A sorted structure has several distinguished subsets (i.e., unary relations) of the universe called "sorts". The "totality" requirement for functions within sorted structures is relaxed in such a way that a function is allowed to be defined on a Cartesian product of various sorts (vs. products of the universe). In addition, within the first order language associated with a sorted structure, each variable x is explicitly attached to a sort S (written x:S) such that x is allowed to assume values only from S. Thus, if a sort consists of a collection of relations over some other sorts, the quantifications over this collection of relations is allowed. In order to differentiate the functions associated with the elements of the signature from other functions constructed within the structure, we'll call the former "built-in functions".

A sorted (*total*) algebra is a structure where all the relations are treated as Boolean-valued functions (Gurevich 1992). (I.e., this would require an explicit sort of Booleans $\mathbb{B}$ = {**true**, **false**}.) For sorted algebras, each variable x *must* be associated with a unique sort, say S (denoted x:S), whereas each constant c *may* be associated with any sort S such that $c \in S$ (similarly, denoted c:S).

A sorted *partial* algebra is a sorted algebra where functions are allowed to be defined on *subsets* of Cartesian products of one or more sorts.

2. SORTED PARTIAL ALGEBRAS WITH EXPLICIT DOMAINS

2.1. Sorted Partial Algebraic First Order Languages with Explicit Domains (SPAED-1-Languages)

We will introduce a new notion of SPAED-1-languages which is a slight modification of standard first order languages.

DEFINITION 2-1 (SPAED-1-Languages) The alphabet of an SPAED-1-language will consist of an algebraic signature, the quantifiers, sorted individual variables, and derivative symbols. Let's define an SPAED-1-language $\mathscr{L}$:

- The algebraic signature of $\mathscr{L}$ is a triple $\Sigma = \langle \mathbf{Sorts}, \mathbf{Func} \rangle$ where:
  - **Sorts** is a finite collection of sorts. Each sort S is intended to denote a set. A sort S is intended to denote a set called the "sort carrier" of S. **Sorts** may be partially ordered via a binary relation "$\leq$" denoting the set inclusion of the sort carriers (i.e., if S, S′ are sorts and $\tilde{S}$, $\tilde{S}'$ are their respective carriers, then if $S \leq S'$ then $\tilde{S} \subseteq \tilde{S}'$. $\mathbf{Sorts} = \mathbf{Sorts}^L \cup \mathbf{Sorts}^N$, where $\mathbf{Sorts}^L$ is the set of "logical" sorts and $\mathbf{Sorts}^N$ is the set of "nonlogical" sorts. $\mathbf{Sorts}^L$ at least includes the sort of Booleans $\mathbb{B}$ and the universal sort $\mathbb{U}$. $\mathbb{U}$ is intended to denote the union of all sort carriers. Each sort S is associated with its characteristic function $(S, (\mathbb{U} \to \mathbb{B}))$, where intuitively, $S(x) = t$ means $x \in S$;
  - **Func** is a finite collection of pairs (f, τ), with τ of the form $(S_1, ..., S_n \to S)$, where $S_1, ..., S_n, S$ are sorts. Intuitively, (f, τ) is intended to denote a function $\tilde{f} : \tilde{S}_1 \times ... \times \tilde{S}_n \to \tilde{S}$, where $\tilde{S}_1, ..., \tilde{S}_n, \tilde{S}$ are the respective sort carriers. We abbreviate $S_1, ..., S_n$ as $^n\vec{S}$ or $\vec{S}$ when there is no confusion and $\tilde{S}_1 \times ... \times \tilde{S}_n$ as $\times^n \vec{S}$ or $\times \vec{S}$. f is called an n-ary function symbol of type τ, τ is called the minimal type of f, $(S_1, ..., S_n)$ are called the argument sorts, and S is called the value sort. If $S_1, ..., S_n \to S$ is the minimal type of f and $S_1 \leq S'_1$, ..., $S_n \leq S'_n$, $S \leq S'$, then we say that f is of type $S'_1, ..., S'_n \to S'$ (abbreviated as $f: S'_1, ..., S'_n \to S'$). Function symbols of arity 0 are called constants; for the constants we sometimes abbreviate $c:(\to S)$ as c:S. $\mathbf{Func} = \mathbf{Func}^L \cup \mathbf{Func}^N$, where $\mathbf{Func}^L$ is the set of "logical" function symbols and $\mathbf{Func}^N$ is the set of "nonlogical" function symbols. $\mathbf{Func}^L$ at least includes the standard Boolean operations $(t, (\to \mathbb{B}))$, $(f, (\to \mathbb{B}))$, $(\neg, (\mathbb{B} \to \mathbb{B}))$ and $(\vee, (\mathbb{B}, \mathbb{B} \to \mathbb{B}))$, etc., as well as the equality on some of the sorts $(=_S, (S, S \to \mathbb{B}))$. It also includes nonstrict Boolean operations introduced later in this section.

- The quantifiers are:
  - strict: $\forall$ and $\exists$;
  - nonstrict: $\overset{ns}{\forall}$ and $\overset{ns}{\exists}$. The meaning of the nonstrict quantifiers will be explained later within the definition of the expression transformer Def.
- Each sorted variable x, in contrast to variables in formal languages without sorts, is explicitly attached to a sort S (written x:S) such that x is allowed to assume values only from S.
- The derivative symbols consist of all symbols of the form Edom.f, where $f:(S_1, ..., S_n \rightarrow S)$ is a functional symbol from **Func**. Edom stands for "explicit domain"; since Edom.f tells for which arguments the function f may be evaluated. Syntactically, we treat Edom.f as a Boolean-valued 2n-ary function symbol of type $(\mathbb{B}, S_1, ..., \mathbb{B}, S_n \rightarrow \mathbb{B})$. However, whereas the function symbol f is intended to denote either a partial or a total function, Edom.f will denote only a total function. Intuitively, the meaning of Edom.f is as follows:
  - if $Edom.f(b_1, t_1, ..., b_n, t_n) = \dot{t}$ then we know the value of $f(t_1, ..., t_n)$ (in other words, $f(t_1, ..., t_n)$ is defined). Moreover, in order to find the value of $f(t_1, ..., t_n)$, we need not know the values of all such t_i where $b_i = \dot{f}$;
  - if $Edom.f(b_1, t_1, ..., b_n, t_n) = \dot{f}$ and $\{i \mid b_i = \dot{f}\} \neq \varnothing$ then, provided that we do not know the values of *all* such t_i where $b_i = \dot{f}$, we also do not know the value of $f(t_1, ..., t_n)$;
  - if $Edom.f(b_1, t_1, ..., b_n, t_n) = \dot{f}$ and $\{i \mid b_i = \dot{f}\} = \varnothing$ then we do not know the value of $f(t_1, ..., t_n)$ (in other words, $f(x_1, ..., x_n)$ is undefined).
  - if Dom.f represents the set of all vectors $(x_1, ..., x_n)$ where the function is defined then $Dom.f = \{(x_1, ..., x_n) \mid Edom.f(\dot{t}, x_1, ..., \dot{t}, x_n) = \dot{t}\}$.
- We will say that $\mathscr{L}$ is an SPAED-1-Language or that $\mathscr{L}$ is a Σ-SPAED-1-Language or, if there is no confusion, that $\mathscr{L}$ is a Σ-Language.

□

Now we'll define the expressions of $\mathscr{L}$.

DEFINITION 2-2 (Expressions and their Sorts) The expressions are defined inductively as follows:

- a variable x:S or a constant c:S are expressions of sort S;
- if $(f, (^n\vec{S} \rightarrow S)) \in$ **Func** and $t_1, ..., t_n$ are expressions (note that we do not require $t_1:S_1, ..., t_n:S_n$) then:
  - $f(t_1, ..., t_n)$ is an expressions of sort S (abbreviated as $f(\vec{t})$);
  - if $b_1:\mathbb{B}, ..., b_n:\mathbb{B}$ are expressions then $Edom.f(b_1,t_1, ..., b_n,t_n)$ is an expression of sort $\mathbb{B}$ (abbreviated as $Edom.f(\vec{b}, \vec{t})$);
- if b is a Boolean-valued expression, then $(\forall x:S, b)$, $(\exists x:S, b)$, $(\overset{ns}{\forall} x:S, b)$, and $(\overset{ns}{\exists} x:S, b)$ are Boolean-valued expressions. Each free occurrence of the variable x:S in t becomes a bound occurrence in $(\forall x:S, b)$, $(\exists x:S, b)$, $(\overset{ns}{\forall} x:S, b)$, and $(\overset{ns}{\exists} x:S, b)$.

To say that e is an expression of sort S, we'll write e:S. We also have the following convention: if $S \subseteq S'$ (i.e., if we can prove $\forall x:\mathbb{U}, S(x) \Rightarrow S'(x)$, see next section) and e:S, then also e:S′. We designate the set of the free variables of e as FV(e).

□

DEFINITION 2-3 (Terms) Terms are expressions without occurrences of explicit domains or quantifiers.

□

2.2. Formal Sorted Partial Algebras with Explicit Domains (FSPED-Algebras)

2.2.1. The Logical Axiom Schema for Explicit Domains

As usual, Boolean expressions are called formulas, and those without bound variables are called closed formulas. The latter may serve as axioms. A formal language with attached axioms is called a formal system. We'll call formal systems based on a SPAED-1-languages formal sorted partial algebras with explicit domains (FSPED-algebras). The axioms attached to an FSPED-algebra consist of logical and nonlogical axioms. The logical axioms include all the usual first order logical axioms for algebraic formal languages (e.g., see [Gries, Schneider 1993]). In addition, logical axioms include the following axioms pertaining to the explicit domain (Edom) notation:

- (A1) (Axioms for all the explicit domains) Let $(f, (^n\vec{S}\rightarrow S))\in$ **Func**, $\{j_1, ..., j_m\}\subseteq\{1, ..., n\}$, $\{i_1, ..., i_k\} = \{1, ..., n\}-\{j_1, ..., j_m\}$, $x_1:S_1, ..., x_n:S_n$, $b_{i_1} = t, ..., b_{i_k} = t$, $b_{j_1} = f, ..., b_{j_m} = f$, and $z_{i_1}:\mathbb{B}, ..., z_{i_k}:\mathbb{B}$. Let us abbreviate $Edom.f((\vec{b}, \vec{x})[z_{i_1}/b_{i_1}, ..., z_{i_k}/b_{i_k}])$ as $Edom.f(\vec{z}_I, \vec{b}_J, \vec{x})$. Then the following is an axiom:
  - $\forall x_{i_1}, ..., x_{i_k}\exists z:\mathbb{B}((\forall x_{j_1}, ..., x_{j_m}, Edom.f(\vec{b}, \vec{x}) = z) \wedge (z = t \Rightarrow \exists y:S\forall x_{j_1}, ..., x_{j_m}, f(\vec{x}) = y) \wedge (z = f \Rightarrow \forall z_{i_1}, ..., z_{i_k}\forall x_{j_1}, ..., x_{j_m}, Edom.f(\vec{z}_I, \vec{b}_J, \vec{x}) = f))$. Intuitively, it means that $Edom.f(\vec{b}, \vec{x})$ does not depend on values of $x_{j_1}, ..., x_{j_m}$, that if $Edom.f(\vec{b}, \vec{x}) = t$ then $f(\vec{x})$ does not depend on values of $x_{j_1}, ..., x_{j_m}$ and that if $Edom.f(\vec{b}, \vec{x}) = f$ then $Edom.f(\vec{z}_I, \vec{b}_J, \vec{x})$ does not depend on either values of $z_{i_1}, ..., z_{i_k}$ or values of $x_{j_1}, ..., x_{j_m}$.

2.2.2. Logical Axioms for Strict Functions

DEFINITION 2-4 (Strict Total Functions) $f:(S_1, ..., S_n\rightarrow S)$ is called a strict total function if $\forall z_1:\mathbb{B}, ..., z_n:\mathbb{B}, x_1:S_1, ..., x_n:S_n(Edom.f(z_1, x_1, ..., z_n, x_n) = (z_1 \wedge ... \wedge z_n))$; (in other words, f is strict total if f is defined when all the arguments are defined.

□

- (A2) (Axioms for Boolean constants)
  - $Edom.t = t$
  - $Edom.f = t$;
- A3) (Axioms for characteristic functions of the form $(S, (\mathbb{U}\rightarrow\mathbb{B}))$)
  - $\forall z:\mathbb{B}, x:\mathbb{U}, Edom.S(z, x) = z$;
- (A3) $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, and $=_S$ are strict total functions. In other words, if $\star:(S, S\rightarrow\mathbb{B})$ is the type declaration of any of $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, and $=_S$, then
  - $\forall z:\mathbb{B}, x:S, w:\mathbb{B}, y:S, Edom.\star(z, x, w, y) = z \wedge w$;

REMARK 2-1 Although we require $Edom.t = t$ and $Edom.f = t$, we will not require that for every constant c, $Edom.c = t$. The advantage of having constants of unknown value will be discussed in the section on skeletons and bundles of SPED-algebras and also in the sections dealing with evolving partial algebras with explicit domains.

□

2.2.3. Logical Axioms for Nonstrict Functions

We will introduce several nonstrict total functions:

- Symmetric nonstrict Boolean: $\wedge_s$, $\vee_s$, $\Rightarrow_s$;
- Right nonstrict Boolean: $\wedge_r$, $\vee_r$, $\Rightarrow_r$;
- Left nonstrict Boolean: $\wedge_l$, $\vee_l$, $\Rightarrow_l$;

- Conditional function (?, (𝔹,𝕌,𝕌→𝕌)) (we'll write (b ? x, y) in lieu of ?(b, x, y)).

Their properties are expressed in the following logical axioms:

- (A5f) As functions, $\wedge_s$, $\vee_s$, $\Rightarrow_s$, $\wedge_r$, $\vee_r$, $\Rightarrow_r$,$\wedge_l$, $\vee_l$, $\Rightarrow_l$ are identical to (respectively) $\wedge$, $\vee$, $\Rightarrow$. I other words, let $\bigstar$:(S, S→𝔹) be any of $\wedge$, $\vee$, or $\Rightarrow$, and let u be any of s, r, or l. Then:
  - $\forall x:\mathbb{B}, y:\mathbb{B}, x \bigstar_u y = x \bigstar y$;
- (A5d) The explicit domains of $\wedge_s$, $\vee_s$, $\Rightarrow_s$, $\wedge_r$, $\vee_r$, $\Rightarrow_r$,$\wedge_l$, $\vee_l$, $\Rightarrow_l$ are defined as follows:
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\vee_s(z, x, w, y) = (z \wedge w) \vee (z \wedge x) \vee (w \wedge y)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\vee_r(z, x, w, y) = (z \wedge w) \vee (z \wedge x)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\vee_l(z, x, w, y) = (z \wedge w) \vee (w \wedge y)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\wedge_s(z, x, w, y) = (z \wedge w) \vee (z \wedge \neg x) \vee (w \wedge \neg y)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\wedge_r(z, x, w, y) = (z \wedge w) \vee (z \wedge \neg x)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\wedge_l(z, x, w, y) = (z \wedge w) \vee (w \wedge \neg y)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\Rightarrow_s(z, x, w, y) = (z \wedge w) \vee (z \wedge \neg x) \vee (w \wedge y)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\Rightarrow_r(z, x, w, y) = (z \wedge w) \vee (z \wedge \neg x)$;
  - $\forall z:\mathbb{B}, x:\mathbb{B}, w:\mathbb{B}, y:\mathbb{B}, Edom.\Rightarrow_l(z, x, w, y) = (z \wedge w) \vee (w \wedge y)$;
- (A6) Conditional function (?, (𝔹,𝕌,𝕌→𝕌)) is defined as follows:

$$(b\ ?\ x, y) \triangleq \begin{cases} x & \text{if } b = \dot{t}; \\ y & \text{if } b = \dot{f}. \end{cases}$$

Note that in order to compute (b ? x, y), when b = $\dot{t}$, we don't have to know the value of y; whereas when b = $\dot{f}$, we don't have to know the value of x. Accordingly, we'll define the explicit domain of ? via the following axiom:

- $\forall z:\mathbb{B}, b:\mathbb{B}, v:\mathbb{B}, x:\mathbb{U}, w:\mathbb{B}, y:\mathbb{U}, Edom.?(z, b, v, x, w, y) \triangleq z \wedge (b \Rightarrow v) \wedge (\neg b \Rightarrow w)$.

Logical axioms are standard in the sense that every FSPED-algebra includes them. In contrast, nonlogical axioms represent properties of functions and sorts of particular FSPED-algebras, e.g., various inclusion relationships on sorts.

2.2.4. Classical First Order Proofs within FSPED-Algebras

In keeping with the Gries idea, we would like the notion of proofs within FSPED-algebras to be the same as for conventional first order theories (with the addition of the logical axioms described above). Recall that, in contrast with conventional first order theories, within FSPED-algebras it is possible to refer during the proof to the value of $f(t_1, ..., t_n)$ when $(t_1, ..., t_n) \notin Dom.f$, which means that $f(t_1, ..., t_n)$ is, in fact, undefined. That wood seem to invalidate the proof. To deal with this problem, Gries used an informal notation Def(e) to denote that the expression e is defined. We'll formalize this approach by giving a rigorous definition of the expression transformer Def via our explicit domain notation.

DEFINITION 2-5 (Expression Transformer Def) For an expression e:S, the expression Def(e) is defined inductively as follows:

- if e is a variable x:S, then Def(e) $\triangleq \dot{t}$;
- if $(f, (S_1, ..., S_m \rightarrow S)) \in$ **Func** and $t_1, ..., t_n$ are expressions then:
  - $Def(f(\vec{t})) \triangleq Edom.f(Def(t_1), t_1, ..., Def(t_n), t_n)$ is an expressions of sort S;
  - if $b_1:\mathbb{B}, ..., b_n:\mathbb{B}$ are expressions, then $Def(Edom.g(\vec{b}, \vec{t})) \triangleq \bigwedge_{i=1}^{n} (Def(b_i) \wedge_s (b_i \Rightarrow Def(t_i)))$;
- if b is a Boolean-valued expression, then
  - $Def(\forall x:S, b) \triangleq \forall x:S, Def(b)$;
  - $Def(\exists x:S, b) \triangleq \forall x:S, Def(b)$;

- $Def(\overset{ns}{\forall}x{:}S, b) \triangleq (\forall x{:}S, Def(b)) \vee \exists x{:}S(Def(b) \wedge_s \neg b)$;
- $Def(\overset{ns}{\exists}x{:}S, b) \triangleq (\forall x{:}S, Def(b)) \vee \exists x{:}S(Def(b) \wedge_s b)$.

□

DEFINITION 2-6 (Applying 1st Order Proofs to Formulas with Partial Functions) In order to prove formulas within FSPED-algebras, we propose the following procedure:

- First, we will use conventional first order proofs (with the addition of the logical axioms described above) in the sense that we will allow within the proofs references to $f(t_1, ..., t_n)$, regardless of $Edom.f(\dot{t},t_1, ..., \dot{t},t_n) = \dot{t}$ or $Edom.f(\dot{t},t_1, ..., \dot{t},t_n) = \dot{f}$. As usual, if a formula φ is derived from the logical axioms only, we write $\vdash \varphi$; if φ is derived using both the logical axioms and a collection Ψ of nonlogical axioms, we write $\Psi \vdash \varphi$ or $\aleph \vdash \varphi$, where $\aleph$ the name of the FSPED-algebra.
- Second, given a closed formula φ, we will try to prove $Def(\varphi)$ (i.e., $Def(\varphi) = \dot{t}$).
- Third, when we have a first order proof that $Def(\varphi) = \dot{t}$, then we'll try to prove φ. If we would fail to prove that $Def(\varphi) = \dot{t}$, then we'll not attempt to prove φ.

□

In the following sections we'll establish the soundness of this approach, i.e., that a formula proved in this way is true on all structures implementing the FSPED-algebra.

DEFINITION 2-7 (Total Closed Expressions) Let e be a closed expression in the language of an FSPED-algebra $\aleph$. If $\aleph \vdash Def(e)$, we'll call e total (with respect to $\aleph$). If $\vdash Def(e)$ then we'll call e logically total.

□

THEOREM 2-1 Def(e) is a logically total formula. In other words, for any expression e, $\vdash Def(Def(e))$.

Proof. By induction on length of expressions.

□

PROPOSITION 2-1 If $b:\mathbb{B}$ then $(\neg Def(b) \vee_s b)$, $(Def(b) \wedge_s b)$, and $(Def(b) \Rightarrow_s b)$ are logically total formulas.

□

2.3. Sorted Partial Algebras with Explicit Domains (SPED-Algebras)

2.3.1. Evaluating Expressions via Interpreting Signatures

DEFINITION 2-8 (SPED-Algebras and Interpreting Signatures) A SPED-algebra is a pair $\mathfrak{A} = \langle \Sigma, \mathfrak{A}[\![\,]\!]\rangle$, where $\Sigma = \langle$**Sorts, Func**$\rangle$ is a signature and $\mathfrak{A}[\![\,]\!]$ is a mapping (called the interpretation of Σ in $\mathfrak{A}$) such that:

- for any sort S, $\mathfrak{A}[\![S]\!]$ is a set called the carrier of S;
- $\mathfrak{A}[\![\mathbb{U}]\!] = \cup_S \mathfrak{A}[\![S]\!]$;
- $\mathfrak{A}[\![\mathbb{B}]\!] = \{\mathfrak{A}[\![\dot{t}]\!], \mathfrak{A}[\![\dot{f}]\!]\}$, where $\mathfrak{A}[\![\dot{t}]\!] \neq \mathfrak{A}[\![\dot{f}]\!]$. For our convenience, we will not distinguish between $\mathbb{B}$ and $\mathfrak{A}[\![\mathbb{B}]\!]$, between $\dot{t}$ and $\mathfrak{A}[\![\dot{t}]\!]$, and between $\dot{f}$ and $\mathfrak{A}[\![\dot{f}]\!]$;
- for any function symbol f with $(f, (^n\vec{S}\rightarrow S)) \in$ **Func**:
  - $\mathfrak{A}[\![f]\!]$ is a function $\mathfrak{A}[\![f]\!]:\mathfrak{A}[\![S_1]\!]\times...\times\mathfrak{A}[\![S_n]\!]\rightarrow\mathfrak{A}[\![S]\!]$ called an operation in $\mathfrak{A}$;
  - $\mathfrak{A}[\![Edom.f]\!]$ is a function $\mathfrak{A}[\![Edom.f]\!]:\mathbb{B}\times\mathfrak{A}[\![S_1]\!]\times...\times\mathbb{B}\times\mathfrak{A}[\![S_n]\!]\rightarrow\mathbb{B}$ called the explicit domain of the operation $\mathfrak{A}[\![f]\!]$;
  - $Dom(\mathfrak{A}[\![f]\!]) = \{\vec{x} \mid x_1 \in S_1 \wedge ... \wedge x_n \in S_n \wedge \mathfrak{A}[\![Edom.f]\!](\dot{t}, x_1, ..., \dot{t}, x_n) = \dot{t}\}$, where Dom(g) is the domain of a partial function g in the usual sense;

- all the logical axioms are satisfied. An adaptation (in terms of SPED-algebras) of the Tarski definition of satisfaction is given below.

□

Now assume that every element of $\mathfrak{A}[\![\mathbb{U}]\!]$ has a name and thus may be considered to be a constant with explicit domain equal to $\mathfrak{t}$. We would like to find values associated with as many expressions of the form $e[a_1/x_1, ..., a_n/x_n]$, where $a_1:S_1, ..., a_n:S_n$ are constants with explicit domain equal to $\mathfrak{t}$ and where $FV(e)\subseteq\{x_1, ..., x_n\}$, as possible. For total algebras such values are computed via the Tarski evaluation. Since we treat our partial functions as total functions with unknown values beyond their explicit domain, we say that the Tarski evaluation of $e[a_1/x_1, ..., a_n/x_n]$ always exists but may be undefined. We designate such values as $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!]$. Below is our modification of Tarski evaluation describing how to find $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!]$ and how to determine whether this value is defined. We will assume that $e:S$ is an expression, $x_1:S_1, ..., x_n:S_n$ are variables, $a_1:S_1, ..., a_n:S_n$ are constants, and that $FV(e)\subseteq\{x_1, ..., x_n\}$.

DEFINITION 2-9 (Evaluation of Expressions) We would like to evaluate $e[a_1/x_1, ..., a_n/x_n]$ and the predicate $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n])$.

- if e is a constant $c:S$, then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{A}[\![c]\!]$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{A}[\![Edom.c]\!]$;
- if e is a variable $x_i:S_i$, then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq a_i$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{t}$;
- if $e = f(\vec{t})$, $(f, (^{m}\vec{S}\rightarrow S))\in$ **Func**, $t_1, ..., t_m$ are expressions, and $\mathfrak{A}[\![t_i[a_1/x_1, ..., a_n/x_n]]\!]\in \mathfrak{A}[\![S_i]\!]$ for i = 1, ..., m, then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{A}[\![f]\!](\mathfrak{A}[\![t_1[a_1/x_1, ..., a_n/x_n]]\!], ..., \mathfrak{A}[\![t_m[a_1/x_1, ..., a_n/x_n]]\!])$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{A}[\![Edom.f]\!](Def^{\mathfrak{A}}(t_1[a_1/x_1, ..., a_n/x_n]), \mathfrak{A}[\![t_1[a_1/x_1, ..., a_n/x_n]]\!], ..., Def^{\mathfrak{A}}(t_m[a_1/x_1, ..., a_n/x_n]), \mathfrak{A}[\![t_m[a_1/x_1, ..., a_n/x_n]]\!])$;
- if $e = \forall x:S, b$ (where $b:\mathbb{B}$ is an expression), then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{t}$ if for every constant $a:S$, $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{t}$;
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{f}$ otherwise;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{t}$ if for every constant $a:S$, $Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{f}$ otherwise;
- if $e = \exists x:S, b$ (where $b:\mathbb{B}$ is an expression), then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{t}$ if for some constant $a:S$, $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{t}$;
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{f}$ otherwise;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{t}$ if for every constant $a:S$, $Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{f}$ otherwise;
- if $e = \overset{ns}{\forall} x:S, b$ (where $b:\mathbb{B}$ is an expression), then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{t}$ if for every constant $a:S$, $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{t}$;
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{f}$ otherwise;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{t}$ if for every constant $a:S$, $Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$ or if for some constant $a:S$, $(Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$ and $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{f})$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{f}$ otherwise;
- if $e = \overset{ns}{\exists} x:S, b$ (where $b:\mathbb{B}$ is an expression), then:
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{t}$ if for some constant $a:S$, $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{t}$;
  - $\mathfrak{A}[\![e[a_1/x_1, ..., a_n/x_n]]\!] \triangleq \mathfrak{f}$ otherwise;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{t}$ if for every constant $a:S$, $Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$ or if for some constant $a:S$, $(Def^{\mathfrak{A}}(b[a_1/x_1, ..., a_n/x_n, a/x]) = \mathfrak{t}$ and $\mathfrak{A}[\![b[a_1/x_1, ..., a_n/x_n, a/x]]\!] = \mathfrak{t})$;
  - $Def^{\mathfrak{A}}(e[a_1/x_1, ..., a_n/x_n]) \triangleq \mathfrak{f}$ otherwise.

□

2.3.2. Models of FSPED-Algebras

DEFINITION 2-10 (Satisfaction of Formulas) We say that a closed formula φ is satisfied by $\mathfrak{A}$ (denoted $\mathfrak{A} \models \varphi$) if $\mathfrak{A}[\![\varphi]\!] = \mathrm{t}$ and $\mathrm{Def}^{\mathfrak{A}}(\varphi) = \mathrm{t}$.

□

Recall that we require that all the logical axioms be satisfied in the above sense.

DEFINITION 2-11 (Models or Implementations of FSPED-Algebras) We say that a SPED-Algebra $\mathfrak{A}$ implements a FSPED-algebra $\aleph$ (or $\mathfrak{A}$ is a model of $\aleph$) if for each nonlogical axiom φ of $\aleph$, $\mathfrak{A} \models \varphi$.

□

THEOREM 2-2 Suppose that a SPED-algebra $\mathfrak{A}$ implements a FSPED-algebra $\aleph$ and that e is a total closed expression in $\aleph$. Then $\mathrm{Def}^{\mathfrak{A}}(e) = \mathrm{t}$.

□

THEOREM 2-3 (Soundness) Suppose that a SPED-algebra $\mathfrak{A}$ implements a FSPED-algebra $\aleph$ and that φ is a total closed formula in $\aleph$ such that $\aleph \vdash \varphi$. Then $\mathfrak{A} \models \varphi$.

□

REMARK 2-2 Sometimes, when there is no confusion, we'll write S instead of $\mathfrak{A}[\![S]\!]$, f instead of $\mathfrak{A}[\![f]\!]$, Edom.f instead of $\mathfrak{A}[\![\mathrm{Edom.f}]\!]$, and Def(e) instead of $\mathrm{Def}^{\mathfrak{A}}(e)$.

□

2.4. Semantics for Partial Functions

2.4.1. Skeletons of SPED-Algebras

DEFINITION 2-12 (Skeletons of SPED-Algebras) Let $\mathfrak{A} = \langle \Sigma, \mathfrak{A}[\![\,]\!] \rangle$ be a SPED-algebra, where $\Sigma = \langle \mathbf{Sorts}, \mathbf{Func} \rangle$. On the basis of $\mathfrak{A}$ we going to construct the following Sped-algebra $\mathfrak{A}^{\perp} = \langle \Sigma^{\perp}, \mathfrak{A}^{\perp}[\![\,]\!] \rangle$, where $\Sigma^{\perp} = \langle \mathbf{Sorts}^{\perp}, \mathbf{Func}^{\perp} \rangle$, which we'll call the skeleton of $\mathfrak{A}$:

- signature $\Sigma^{\perp}$:
  - $\perp$ is a new function symbol (i.e., $\perp \notin \mathbf{Func}$);
  - $\mathbf{Sorts}^{\perp} \triangleq \mathbf{Sorts} \cup \{S^{\perp} \mid S \in \mathbf{Sorts}\}$;
  - $\mathbb{U}^{\perp}$ is the universe of $\Sigma^{\perp}$;
  - $\mathbf{Func}^{\perp} \triangleq \{(f^{\perp}, (S^{\perp}_1, ..., S^{\perp}_m \rightarrow S^{\perp})) \mid (f, (S_1, ..., S_m \rightarrow S)) \in \mathbf{Func}\} \cup \{(\perp, (\rightarrow \mathbb{U}^{\perp}))\}$;
- the interpretation map $\mathfrak{A}^{\perp}[\![\,]\!]$ on constant $\perp$ and its explicit domain:
  - $\mathfrak{A}^{\perp}[\![\perp]\!] \notin \mathfrak{A}[\![\mathbb{U}]\!]$. From now on we will not distinguish between $\perp$ and $\mathfrak{A}^{\perp}[\![\perp]\!]$;
  - $\mathfrak{A}^{\perp}[\![\mathrm{Edom.}\perp]\!] \triangleq \mathrm{f}$;
- the interpretation map $\mathfrak{A}^{\perp}[\![\,]\!]$ on sorts:
  - for each $S \in \mathbf{Sorts}$, $\mathfrak{A}^{\perp}[\![S]\!] \triangleq \mathfrak{A}[\![S]\!]$;
  - for each $S \in \mathbf{Sorts}$, $\mathfrak{A}^{\perp}[\![S^{\perp}]\!] \triangleq \mathfrak{A}[\![S]\!] \cup \{\perp\}$;
- the interpretation map $\mathfrak{A}^{\perp}[\![\,]\!]$ on function symbols and explicit domains:
  (we abbreviate $z_1, x_1, ..., z_n, x_n$ as $\vec{z}, \vec{x}$ and $x_1, ..., x_n$ as $\vec{x}$)
  - on function symbols:

    for each $(f, (S_1, ..., S_n \rightarrow S)) \in \mathbf{Func}$, for all $x_1 \in \mathfrak{A}^{\perp}[\![S^{\perp}_1]\!]$, ..., $x_n \in \mathfrak{A}^{\perp}[\![S^{\perp}_n]\!]$, if $\{ j_1, ..., j_m\} = \{i \mid x_i = \perp\}$ and $z_1 = \mathrm{t}$, ..., $z_n = \mathrm{t}$, then

$$\mathfrak{A}^{\perp}[\![f^{\perp}]\!](\vec{x}) \triangleq \begin{cases} \mathfrak{A}[\![f]\!](\vec{x}'), & \text{if } \mathfrak{A}[\![\text{Edom.f}]\!](\vec{z}', \vec{x}') = t, \text{ where } \vec{z}' = \vec{z}[\dot{f}/z_{j_1}, ..., \dot{f}/z_{j_m}] \text{ and } \vec{x}' = \\ & \vec{x}[y_{j_1}/x_{j_1}, ..., y_{j_m}/x_{j_m}] \text{ for arbitrary } y_{j_1} \in \mathfrak{A}[\![S_{j_1}]\!], ..., y_{j_m} \in \mathfrak{A}[\![S_{j_m}]\!], \\ \perp, & \text{otherwise.} \end{cases}$$

- on explicit domains:

  for each $(f, (S_1, ..., S_n \to S)) \in \mathbf{Func}$, $z_1 \in \mathbb{B}$, $x_1 \in \mathfrak{A}^{\perp}[\![S^{\perp}{}_1]\!]$, ..., $z_n \in \mathbb{B}$, $x_n \in \mathfrak{A}^{\perp}[\![S^{\perp}{}_n]\!]$, if $\{j_1, ..., j_m\} = \{i \mid z_i = \dot{f}\}$ then

$$\mathfrak{A}^{\perp}[\![\text{Edom.f}^{\perp}]\!](\vec{z}, \vec{x}) \triangleq \begin{cases} \dot{f} & \text{if for some } i \in [1..n],\ z_i = t \text{ and } x_i = \perp; \\ \mathfrak{A}[\![\text{Edom.f}]\!](\vec{z}, \vec{x}') & \text{otherwise, where } = \vec{x}' = \vec{x}[y_{j_1}/x_{j_1}, ..., y_{j_m}/x_{j_m}] \\ & \text{for arbitrary } y_{j_1} \in \mathfrak{A}[\![S_{j_1}]\!], ..., y_{j_m} \in \mathfrak{A}[\![S_{j_m}]\!]. \end{cases}$$

□

COROLLARY. $\mathbb{U}^{\perp}$ is a SPED-algebra.

□

THEOREM 2-4 Let φ closed formula in the language of $\mathfrak{A}$, such that $\text{Def}(\varphi) = t$. Then φ is satisfied in $\mathfrak{A}$ if and only if φ is satisfied in $\mathfrak{A}^{\perp}$.

□

2.4.2. Bundles of SPED-Algebras

DEFINITION 2-13 (Bundles of SPED-Algebras) Let $\mathfrak{A} = \langle \Sigma, \mathfrak{A}[\![\,]\!] \rangle$ be a SPED-algebra, where $\Sigma = \langle \mathbf{Sorts}, \mathbf{Func} \rangle$ and let $\perp \notin \mathfrak{A}[\![\mathbb{U}]\!]$. We call the class of all such SPED-algebras $\mathfrak{A}' = \langle \Sigma, \mathfrak{A}'[\![\,]\!] \rangle$ where $\mathfrak{A}'^{\perp}$ is identical with $\mathfrak{A}^{\perp}$, a bundle of SPED-algebras.

□

COROLLARY. All the SPED-algebras in a bundle satisfy the same set of total statements.

□

REMARK 2-3 (Bundles, Skeletons, and the original D. Gries Idea) Since a bundle of SPED-algebras contains all possible algebras where the values of functions outside their domains defined in every which way, the bundle embodies the idea that a partial function may be thought of as a total function such that the values outside its domain somehow exist, but are unknown. On the other hand, the skeletons represent traditional treatment of partiality via 3-valued logic. We have just shown that they coincide.

3. SEMANTICS OF PROGRAM CORRECTNESS

3.1. Semantics of Programs with Partial Operations via Evolving Sorted Partial Algebras with Explicit Domains (ESPED-Algebras)

Given a specification, our intuitive concept of a program satisfying this specification is a state machine transforming the states defined by the data structure of the specification. Although there many descriptions of formal program semantics (Harel 1979; and Loeckx, Sieber 1987, etc., etc.), the most convenient for us is the "evolving algebras" semantics developed by Y. Gurevich (Gurevich 1993). We'll modify the original evolving algebras to accommodate our explicit domains.

DEFINITION 3-1 (ESPED-Signature) ESED-signature is a triple $\Sigma_e = \langle \mathbf{Sorts}, \mathbf{Func}_s, \mathbf{Func}_d \rangle$, such that $\Sigma_d = \langle \mathbf{Sorts}, \mathbf{Func}_s \cup \mathbf{Func}_d \rangle$ is a signature as before and $\mathbf{Func}_s$ and $\mathbf{Func}_d$ are disjoint. We call $\mathbf{Func}_s$ the set of static function symbols and $\mathbf{Func}_d$ the set of dynamic function symbols. We'll call the dynamic

function symbols program variables. The program variables of 0 arity are called not indexed and the rest of elements of $\mathbf{Func}_d$ are called indexed.

□

DEFINITION 3-2 (ESPED-Algebras) An ESPED-algebra is a quintuple $\mathfrak{E} = \langle \Sigma_e, SV, \mathfrak{B}, \mathfrak{S}, \mathscr{P} \rangle$, where $\Sigma_e = \langle \mathbf{Sorts}, \mathbf{Func}_s, \mathbf{Func}_d \rangle$ is a ESPED-signature, SV is a subset of $\mathbf{Func}_d$ called the list of specification variables, $\mathfrak{B} = \langle \Sigma_s, \mathfrak{B}[\![]\!] \rangle$ SPED-algebra (where $\Sigma_s = \langle \mathbf{Sorts}, \mathbf{Func}_s \rangle$), $\mathfrak{S} = \{ \mathfrak{A} = \langle \Sigma_d, \mathfrak{A}[\![]\!] \rangle \mid \mathfrak{A}|_{\Sigma_s} = \mathfrak{B} \}$ (where $\Sigma_d = \langle \mathbf{Sorts}, \mathbf{Func}_s \cup \mathbf{Func}_d \rangle$ and $\mathfrak{A}|_{\Sigma_s}$ is a restriction of $\mathfrak{A}$ to Σ_s), and $\mathscr{P}$ is a algorithm on $\mathfrak{S}$ (we define algorithms below). We call $\mathfrak{B}$ the base algebra and $\mathfrak{S}$ the set of program states or superuniverse (due to Gurevich).

$\mathfrak{E}$ may be thought of as a state transition system (STS). Given an initial state from the superuniverse, it will commence a series of state jumps producing a *run*. The state jumps are governed by the algorithm $\mathscr{P}$. The additional restrictions on the state runs is that the specification variables must not be explicitly present in the algorithms and thus must not be changed during the state jumps (see remark below).

□

REMARK 3-1 Note that for a program variable f it is possible that in a given initial state $\mathfrak{A}$, the value of $\mathfrak{A}[\![\mathrm{Edom.f}]\!](z_1, x_1, \ldots, z_n, x_n)$, where $n \geq 0$, would be equal to $\dagger$ for all arguments. This provides a semantics to the notion of "uninitialized program variables". In the following sections the question of correctness of sequential programs with uninitialized program variables is completely solved.

Now about syntactic variables. Although the their original treatment [Gries 1981, Morgan 1991, etc.] assumed that their values should not be changed, there are cases when they could be thought as functions of the program state and thus may be changed without explicitly appearing in a program. Thus in [Yakhnis, Farrell, Shultz 1994] the syntactic variables are subdivided into static syntactic variables and dynamic syntactic variables. The latter are beyond the scope of this discourse.

A formal definition of STS capable of producing either finite or infinite runs is given in [Yakhnis, Stilman 1994, 1995]. Here we'll limit our discussion only to finite runs of ESPED-algebras representing sequential programs. We'll discuss concurrent and/or infinitely running ESPED-algebras in [A. Yakhnis, V. Yakhnis, Semantics of Concurrent Communicating Objects, in preparation].

□

DEFINITION 3-3 (Semantics of Programs) A semantics for a program in most programming language is an ESPED-algebra assigned in a natural way.

□

We would like to construct algorithms using as a base five simplified constructs from the Dijkstra language. Living the Skip and Composition instructions intact, we modified the Assignments, IF, and Loop via taking advantage of our expressions transformer Def. For ease in provability, we incorporated the invariant and the bound function directly within the Simple Loop (as was done in [Yakhnis, Farrell, Shultz 1994]). Also, we added a modified form of the Pseudocode Instruction from [Yakhnis, Farrell, Shultz 1994]. Although the algorithm behavior below is described intuitively, it is quite easy to formalize using the machinery developed above. E.g., "compute the value of Def(E) in the initial program state" means "on the basis of DEFINITION 2-9 find the result of evaluation of $\mathrm{Def}^{\mathfrak{A}}(E)$ in respect to the algebra $\mathfrak{A}$ representing the initial state".

Instructions	Behavior during Execution
Skip ***skip***	Step 1. Do nothing; Step 2. Terminate.
Composition /* $\mathscr{F}$ and $\mathscr{G}$ are algorithms */ $\mathscr{F}; \mathscr{G}$	Step 1. Execute $\mathscr{F}$; Step 2. Execute $\mathscr{G}$; Step 3. Terminate.

Simple Assignment /* x:S is a variable and E:S is an expression */ x := E	Step 1. Compute the value of Def(E) in the initial program state. If Def(E) = ↯ then crash. Otherwise go to the next step; Step 2. Get the new program state by replacing the value of the program variable x by the value of E, replacing Edom.x by ↯, and leaving the values of all other variables unchanged; Step 3. Terminate.
Strict Indexed Assignment /* let $f:S_1, ..., S_n \to S$ be an indexed program variable, $t_1:S_1, ..., t_n:S_n$, E:S be expressions */ $f(t_1, ..., t_n) := E$	Step 1. Compute the values of $Def(t_1)$, ..., $Def(t_n)$, Def(E) in the initial program state. If any is equal to ↯ then crash. Otherwise go to the next step; Step 2. Get the new program state by replacing the value of $f(t_1, ..., t_n)$ by the value of E, replacing the value of Edom.$f(↯, t_1, ..., ↯, t_n)$ by ↯ and leaving the values of all other variables unchanged; Step 3. Terminate.
Simple IF /* γ is a Boolean expression and $\mathscr{F}$ and $\mathscr{G}$ are algorithms. */ **if** $\gamma \to \mathscr{F}$ $\square \neg\gamma \to \mathscr{G}$ **fi**	Step 1. Evaluate Def(γ) in the initial program state. If Def(γ) = ↯ then crash. Otherwise go to the next step; Step 2. If γ it evaluates as t, execute $\mathscr{F}$. Otherwise execute $\mathscr{G}$; Step 3. Terminate.
Simple Verifiable Loop /* γ is a Boolean expression, φ is a logical assertion, E is an integer-valued specification expression and $\mathscr{F}$ is an algorithm. It is established that φ is an invariant of $\mathscr{F}$ and that E is a bound function. */ **do** $\gamma \to$ **invariant** φ **bound function** E $\mathscr{F}$ **od**	/* The following must be proved beforehand: • $\{\varphi \wedge_s \gamma\}\ \mathscr{F}\ \{\varphi\} \wedge_s (Def(\varphi) \wedge_s \varphi \Rightarrow_s Def(\gamma))$, i.e., φ is an invariant of the loop; • $(\varphi \Rightarrow_s E \geq 0) \wedge_s \{E=X\}\ \mathscr{F}\ \{E<X\}$, where X is an integer program variable not occurring in $\mathscr{F}$. Thus E is a bound function of the loop. */ Step 1. Evaluate Def(φ) in the initial program state. If Def(φ) = ↯, then crash. Otherwise go to step 2; Step 2. Evaluate the loop guard γ. If γ evaluates as ↯, then terminate. Otherwise go to step 3; Step 3. Execute the loop body $\mathscr{F}$. When and if $\mathscr{F}$ terminates, go to step 2.
Pseudocode Instruction /* SV is a list of program variables called "specification variables", φ, ψ are logical assertions */ I[SV, φ, ψ]I	Step 1. Evaluate Def(φ) in the initial program state. If Def(φ) = ↯, or if Def(φ) = ↓ and φ = ↯ then crash. Otherwise go to step 2; Step 2. Let Φ be the set of all program states such that: • the values of all the specification variables are the same as in the initial state; • the state satisfies ψ. If Φ is not empty then choose any state from Φ as the final state and terminate. Otherwise crash.

REMARK 3-2 There are two other kinds of assignments, **Strict Indexed Subspace Assignment** and **Nontrict Indexed Assignment**. The former assigns a single value to a subspace of the argument space of an indexed program variable, whereas the latter permits to assign a value when some of the arguments are undefined. We believe that these provable constructs would increase the expressive power of the modern programming languages.

□

3.2. Extending Dijkstra-Gries Program Correctness Rules to Programs with Partial Operations

Using our formalization of the Gries idea, we will first extend the Dijkstra weakest precondition (wp) expression transformer to programs with partial operations. We'll denote the new transformer wpp for "weakest precondition with partiality". We assume that Q is a logical assertion; the rest of the symbols are from the above semantic definitions.

Instruction $\mathscr{P}$	wpp($\mathscr{P}$, Q) $\triangleq$
Skip ***skip***	• Def(Q) $\wedge_s$ Q
Composition $\mathscr{F}$; $\mathscr{G}$	• wpp($\mathscr{F}$; wpp($\mathscr{G}$, Q))
Simple Assignment x := E	• Def(E) $\wedge_s$ Def(Q[E/x]) $\wedge_s$ Q[E/x]
Strict Indexed Assignment $f(t_1, ..., t_n) := E$	Let $f[t_1, ..., t_n \mapsto E]$ be a function identical to f, except that $f(t_1, ..., t_n) = E$. Then: • wpp($f(t_1, ..., t_n) := E$, Q) $\triangleq$ Def(t_1) $\wedge_s$ ... $\wedge_s$ Def(t_n) $\wedge_s$ Def(E) $\wedge_s$ Def(Q[$f[t_1, ..., t_n \mapsto E]$/f]) $\wedge_s$ Q[$f[t_1, ..., t_n \mapsto E]$/f]
Simple IF **if** $\gamma \rightarrow \mathscr{F}$ $\square \neg\gamma \rightarrow \mathscr{G}$ **fi**	• Def(γ) $\wedge_s$ ($\gamma \Rightarrow$ wpp($\mathscr{F}$, Q)) $\wedge_s$ ($\neg\gamma \Rightarrow$ wpp($\mathscr{G}$, Q))
Simple Verifiable Loop **do** $\gamma \rightarrow$ **invariant** φ **bound function** E $\mathscr{F}$ **od**	• φ if Def(φ) $\wedge_s$ $\varphi \wedge \neg\gamma \Rightarrow$ Def(Q) $\wedge_s$ Q; • $\dagger$ otherwise.
Pseudocode Instruction $\lvert[SV, \varphi, \psi]\rvert$	• φ if Def(φ) $\wedge_s$ $\varphi \Rightarrow$ Def(Q) $\wedge_s$ Q; • $\dagger$ otherwise.

3.3. Hoare Triples With Partial Functions

We consider Hoare triples $\{P\}\mathscr{R}\{Q\}$ where the assertions P (the "precondition") and Q (the "postcondition") and the program $\mathscr{R}$ may have occurrences of partial functions and also where the program $\mathscr{R}$ may have uninitialized program variables. We limit ourselves to discussing the "total correctness" semantics of Hoare triples since within safety-critical systems the "partial correctness" semantics is of limited value.

We will treat $\mathscr{R}$ as an ESPED-algebra and we'll assume that $\mathscr{R}$ describes all the symbols from P and Q. Now, within the total correctness semantics we say that $\{P\}\mathscr{R}\{Q\}$ holds if the following is true:

- for each initial state satisfying Def(P) $\wedge_s$ P the program run will posses the following qualities:
  - it would be finite;
  - for every state in the run the values of the syntactic variables would be identical with those on the initial state;

- during each state transition there would be no attempt to evaluate a function outside its current domain;
- the final state would satisfy Def(P) $\wedge_s$ P.

THEOREM 3-1 (Total Correctness Semantics of Hoare Triples) If Def(P) $\wedge_s$ P $\Rightarrow$ wpp($\mathscr{R}$, Q) evaluates as $\mathfrak{t}$ on each SPED-algebra representing an initial state, then {P}.$\mathscr{R}${Q} holds.
Proof. Induction on the length of the algorithms. □

COROLLARY If all the initial states are models of a an FSPED-algebra $\aleph$, then if $\aleph \vdash$ Def(P) $\wedge_s$ P $\Rightarrow$ wpp($\mathscr{R}$, Q) then {P}.$\mathscr{R}${Q} holds. □

The last theorem and its corollary provide a solid foundation for automating proofs of safety-critical systems.

4. CONCLUSION

4.1. What We Have Achieved

- Formalized the original Gries idea about proofs of program correctness with partial functions. Thus various works on program derivation [Gries 1982, Kaldewaij 1990, Morgan 1991, etc.] are extended in the ream of partial functions.
- Formalized the notions of functions with argument lists of variable length. Thus such languages as C/C++ would be able to enter in the realm of program correctness proofs.
- Provided a solid foundation for automating proofs of safety-critical systems.

4.2. Our Future Work

- In [A. Yakhnis, V. Yakhnis, Semantics of Concurrent Communicating Objects, in preparation] we'll provide semantics of correctness proofs of:
  - concurrent software;
  - perpetually running software;
  - object classes with partial operations;
  - communicating truly concurrent objects.
- In [A. Yakhnis, V. Yakhnis, First-Order Basis for Automated Checking of Software Build from Partial and Nondeterministic Operations, to be submitted to CADE-13 Workshop on Mechanization Of Partial Functions, July 30, 1996] we'll provide the following features of our methodology (which were omitted from the present paper because of space limitation):
  - precise semantics of the stepwise refinement process with partial operations via homomorphisms of SPED-algebras;
  - partial nondeterministic functions;
  - correctness of generic algorithms with partial functions and in the presence of dynamic specification variables.

ACKNOWLEDGMENTS

We would like to express our gratitude to Anil Nerode for his breadth of knowledge in logic and computer science that he was instilling in us during our stay at Cornell. We are grateful to David Gries for teaching us the program derivation. Many thanks to Larry Dalton, our manager at Sandia National Labs, without whose leadership of Sandia's High Integrity Software Initiative, common sense and keen engineering intuition this work would not be possible.

ABOUT THE AUTHORS

Dr. Alexander R. Yakhnis is a member of Command and Control Software Department (org. 2615) at Sandia National Laboratories. He received a Diploma in Mathematics from Moscow State University, Moscow, Russia in 1973. He worked as a computer programmer in Moscow, Russia and Houston,

Texas. Dr. A. Yakhnis received an M.S. in Computer Science and a Ph.D. in Mathematics/Computer Science from Cornell University, Ithaca, New York in 1990. His research was in program correctness for concurrent and sequential programs, winning strategies for two person games, control theory, hybrid systems, and object-oriented methods. Dr. A. Yakhnis worked as a Research Scientist at Mathematical Sciences Institute, Cornell University until June 1995. He joined Sandia National Laboratories at Albuquerque in July 1995. He can be reached by phone at (505) 844-0277 or by e-mail at aryakhn@sandia.gov.

Dr. Vladimir R. Yakhnis is a member of Command and Control Software Department (org. 2615) at Sandia National Laboratories. He received a Diploma in Mathematics from Moscow State University, Moscow, Russia in 1975. He worked as a computer programmer in Moscow, Russia and Houston, Texas. Dr. V. Yakhnis received an M.S. in Computer Science and a Ph.D. in Mathematics/Computer Science from Cornell University, Ithaca, New York in 1990. His research was in program correctness for concurrent and sequential programs, winning strategies for two person games, state transition systems and object-oriented methods. Dr. V. Yakhnis worked at the IBM Endicott Programming Laboratory as an Advisory Programmer until 1994. He worked as a Visiting Scientist at Mathematical Sciences Institute, Cornell University until June 1995. He joined Sandia National Laboratories at Albuquerque in July 1995. He can be reached by phone at (505) 844-8672 or by e-mail at vryakhn@sandia.gov.

REFERENCES

Apt, K. R. (1981) Ten Years of Hoare's Logic, a Survey," *ACM Trans. on Prog. Lang. and Sys.*, 3, 431-483, 1981.

Apt, K. R., Olderog, E. R. (1991) *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.

Bohorquez, J., Cardoso, R. (1993) Problem Solving Strategies for the Derivation of Programs, *Logical Methods* (J. N. Crossley et al, ed.), Birkhauser, 1993.

Breu, R., *Algebraic Specification Techniques in Object Oriented Programming Environments*, Springer-Verlag, 1991.

Burmeister, P. (1986) A Model Theoretic Oriented Approach to Partial Algebras, *Mathematical research 31*, Akademie-Verlag, Berlin , 1986

Burstall, R.M., Goguen, J.A. (1977) Putting Theories together to Make Specifications, in *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 1045-1058, 1977.

Cohen, E., *Programming in the 90s: An Introduction to the Calculation of Programs*, Springer-Verlag, 1990.

Dijkstra, E.W., *A Discipline of Programming*, Prentice Hall, 1976.

Dijkstra, E.W., Feijen, W.H.J., *A Method of Programming*, Addison Wesley, 1988.

Dijkstra, E.W., Scholten, C.S., *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.

Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

Gumb, D., *Programming Logics*, John Wiley & Sons, 1989.

Gurevich, Y. (1995) Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, pp. 7-36, Oxford University Press, 1995.

Guttag, J.V., Horning, J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

Harel, D., *First-Order Dynamic Logic*, Springer-Verlag, 1979.

Hoare, C.A.R., "An Axiomatic Approach to Computer Programming," in *Essays in Computer Science*, C. A. .R. Hoare and C. B. Jones (eds), Prentice-Hall, 1989.

Hopcroft, J., Ullman, J. (1979) *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 1979.

Jones, C.B., *Systematic Software Development using VDM*, Prentice-Hall International 1990.

Kaldewaij, A., *Programming: The Derivation of Algorithms*, Prentice Hall, 1990.

Kieburtz, R.B., Shultis, J. (1981) Transformations of FP Program Schemes, *Proceedings of the Conference on Functional Programming and Architecture*, pp. 41-48, 1981.

Loeckx, J., Sieber, K., *The Foundations of Program Verification*, John Wiley & Sons, 1987.

Morgan, C., *Programming from specifications*, Oxford University Press, 1991.

Nerode, A., Remmel, J. B., Yakhnis, A. (1993) Hybrid System Games: Extraction of Control Automata with Small Topologies, Mathematical Sciences Institute, Cornell University, Technical Report 93-102, 61 pp., 1993.

Nerode, A., Remmel, J. B., Yakhnis, A. (1995) Controllers as Fixed Points of Set-Valued Operators, *International Conference on Intelligent Control*, Monterey, California, 1995.

Nerode, A., Remmel, J. B., Yakhnis, A. (1995) Differential Inclusions and Fixed POints for Design and Verification of Hybrid Systems, *Hybrid Systems II, Lecture Notes in Computer Science #999*, pp. 344-358, Springer, 1995.

Nerode, A., Yakhnis, A. (1992) Modeling Hybrid Systems as Games, *Proceedings of the 31st IEEE Conference on Decision and Control*, pp. 2947-2952, 1992.

Nerode, A., Yakhnis, A., Yakhnis, V. (1992) Concurrent Programs as Strategies in Games, in *Logic From Computer Science*, MSRI series (Y. Moschovakis, ed.), pp. 405-479, Springer-Verlag, 1992.

Nerode, A., Yakhnis, A., Yakhnis, V. (1993) Distributed Concurrent Programs as Strategies in Games, in *Logical Methods* (J. N. Crossley et al, ed.), pp. 624-653, Birkhauser, 1993.

Owicki, S., Gries, D., An Axiomatic Proof Technique for Parallel Programs, *Acta Informatica*, No. 6, pp. 319-340, 1976.

Reichel, H. (1987) *Initial computability, Algebraic Specifications, and Partial Algebras*, Claredon Press, Oxford, 1987

Stilman, B. (1995b) Multiagent Air Combat with Concurrent Motions, Symposium on Linguistic Geometry and Semantic Control, *Proc. of the First World Congress on Intelligent Manufacturing: Processes and Systems,* Mayaguez, Puerto Rico, Feb. 1995.

Tucker, J.V., Zucker, J.I., *Program Correctness over Abstract Data Types with Error-State Semantics*, North-Holland, 1988.

Wirsing, M., "Algebraic Specification," in *Handbook of Theoretical Computer Science*, pp. 675-788, Elsevier Science Publishers B.V., 1990.

Woodcock, J., Loomes, M., *Software Engineering Mathematics*, Pitman, 1988.

Woodcock, J.C.P., "The Rudiments of Algorithm Refinement," The Computer Journal, vol. 35, num. 5, October 1992.

Yakhnis, A. (1989) Concurrent Specifications and their Gurevich-Harrington Games and Representation of Programs as Strategies, *Transactions of the 7th (June 1989) Army Conference on Applied Mathematics and Computing*, pp. 319-332, 1990.

Yakhnis, A., Yakhnis, V. (1990) Extension of Gurevich-Harrington's Restricted Memory Determinacy Theorem: a Criterion for the Winning Player and an Explicit Class of Winning Strategies, *Annals of Pure and Applied Logic*, Vol. 48, pp. 277-297, 1990.

Yakhnis, A., Yakhnis, V. (1993) Gurevich-Harrington's Games Defined by Finite Automata, *Annals of Pure and Applied Logic*, Vol. 62, pp. 265-294, 1993.

Yakhnis, A., Yakhnis, V., First-Order Basis for Automated Checking of Software Build from Partial and Nondeterministic Operations, to be submitted to *CADE-13 Workshop on Mechanization Of Partial Functions*, July 30, 1996.

Yakhnis, A., Yakhnis, V., Semantics of Concurrent Communicating Objects, in preparation.

Yakhnis, V. (1989) Extraction of Concurrent Programs from Gurevich-Harrington Games, *Transactions of the 7th (June1989) Army Conference on Applied Mathematics and Computing*, pp.333-343, 1990.

Yakhnis, V., Farrell, J., Shultz, S. (1994) Deriving Programs Using Generic Algorithms, *IBM Systems Journal*, vol. 33, no. 1, pp. 158-181, 1994.

Yakhnis, V., Stilman, B. (1995a) Foundations of Linguistic Geometry: Complex Systems and Winning Conditions, *Proceedings of the First World Congress on Intelligent Manufacturing Processes and Systems (IMP&S)*, February 1995.

Yakhnis, V., Stilman, B. (1995b) A Multi-Agent Graph-Game Approach to Theoretical Foundations of Linguistic Geometry, *WOCFAI 95*, Paris, 3-7 July 1995.