Software with Partial Functions: Automating Correctness Proofs via Nonstrict Explicit Domains*

Alexander Yakhnis
Command and Control Software, org. 2615, Sandia National Laboratories, MS-0535, 1515 Eubank SE, Albuquerque, NM 87123, aryakhn@sandia.gov

Vladimir Yakhnis
Command and Control Software, org. 2615, Sandia National Laboratories, MS-0535, 1515 Eubank SE, Albuquerque, NM 87123, vryakhn@sandia.gov

Victor Winter
Advanced Engineering & MFG Software Development, org. 9622, Sandia National Laboratories, MS-0660, 1515 Eubank SE, Albuquerque, NM 87123, vlwinte@sandia.gov

CADE-13 Workshop on Mechanization of Partial Functions

New Brunswick, Rutgers University, 30 July 1996

* This work was supported by the United States Department of Energy under contract DE-AC04-94AL84000.

DISCLAIMER

Software with Partial Functions: Automating Correctness Proofs via Nonstrict Explicit Domains*

Alexander Yakhnis
Command and Control Software, org. 2615, Sandia National Laboratories, MS-0535, 1515 Eubank SE, Albuquerque, NM 87123, aryakhn@sandia.gov

Vladimir Yakhnis
Command and Control Software, org. 2615, Sandia National Laboratories, MS-0535, 1515 Eubank SE, Albuquerque, NM 87123, vryakhn@sandia.gov

Victor Winter
Advanced Engineering & MFG Software Development, org. 9622, Sandia National Laboratories, MS-0660, 1515 Eubank SE, Albuquerque, NM 87123, vlwinte@sandia.gov

ABSTRACT As our society becomes technologically more complex, computers are being used in greater and greater numbers of high consequence systems. Giving a machine control over the lives of humans can be disturbing, especially if the software that is run on such a machine has bugs. Formal reasoning is one of the most powerful techniques available to demonstrate the correctness of a piece of software.

When reasoning about software and its development, one frequently encounters expressions that contain partial functions. As might be expected, the presence of partial functions introduces an additional dimension of difficulty to the reasoning framework. This difficulty produces an especially strong impact in the case of high consequence systems.

An ability to use formal methods for constructing software is essential if we want to obtain greater confidence in such systems through formal reasoning. This is only reasonable under automation of software development and verification. However, the ubiquitous presence of partial functions prevents a uniform application to software of any tools not specifically accounting for partial functions.

In this paper we will describe a framework for reasoning about software, based on the *nonstrict explicit domain approach* [17, 18], that is applicable to a large class of software/hardware systems. In this framework the Hoare triples containing partial functions can be reasoned about automatically in a well-defined and uniform manner.

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

1. INTRODUCTION

1.1. Motivation

When constructing high consequence software/hardware systems, it is essential that one has the ability to reason about properties of computations expressed by Hoare triples. Recall that the Hoare triple $\{P\}c\{R\}$ states that if the code fragment c begins its execution at any state s_0 such that $P(s_0)$ holds, then

- c terminates
- upon termination, c produces a state s_1 such that $R(s_1)$ holds.

Although the Hoare triples and their proof rules [3] are the standard mechanism for proving the correctness of terminating programs, the current state of the correctness proof practice does not adequately address Hoare triples that contain partial functions.

Partial functions were dealt with in mathematics rigorously for quite a long time [10]. However, with respect to software there is more difficulty in handling partial functions. 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 is a common occurrence within software. 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. Using our new notion of "nonstrict explicit domains", we alleviate the above obstacles, thus providing a uniform and practical way for proving correctness of software with partial functions. In particular, nonstrict explicit domains permit us to utilize the existing theorem provers to verify software under conditions when software and/or its requirements contain partial functions.

* This work was supported by the United States Department of Energy under contract DE-AC04-94AL84000.

1.2. The Goals of the Paper

1.2.1. Extending Logical Connectors

Consider a typical high consequence system: a nuclear reactor, e.g., the EBR II sodium reactor. The nuclear core of this reactor is cooled by liquid sodium. Now suppose that it has been determined that this reactor should be shut down if the ratio of the heat to the coolant flow exceeds a certain threshold T. This shutdown condition can be expressed as:

$$heat/coolant\_flow \geq T \Rightarrow reactor\_shutdown$$

Here *reactor_shutdown* is the property of the reactor to be in a shutdown state. We assume that there is a command *shutdown* satisfying the following Hoare triple $\{true\}shutdown\{reactor\_shutdown\}$. In this context, consider the possibility that the flow that cools the reactor can stop (i.e., $coolant\_flow = 0$) due to some mechanical failure. Clearly, one would also like to shutdown the reactor in this case:

$$coolant\_flow = 0 \Rightarrow reactor\_shutdown$$

If software is supposed to control the shutdown in those two cases, the following Hoare triple for a C-code fragment might be written:

Precondition: $\{heat/coolant\_flow \geq T \vee coolant\_flow = 0\}$

if *(coolant_flow = 0 || heat/coolant_flow ≥ T) shutdown* /* where || is the C notation for "OR" */

Postcondition: *{reactor_shutdown}*

One of the standard approaches to proving correctness of the above Hoare triple is to show that

$$Precondition \Rightarrow \text{wp}(c, Postcondition)$$

holds, where c is the above code fragment. However, in such an approach, a difficulty is introduced by the presence of partial functions, which can occur in either the pre/postconditions or in the code itself, e.g., in the above example the division "/" occurs both in the precondition and the code. Intuitively, the code fragment in the above example is correct. Nevertheless, showing within a standard two valued logic that the program operates correctly when the initial state satisfy $coolant\_flow = 0$ presents problems. In this particular case one could rewrite the precondition to

$$Precondition': \{heat \geq coolant\_flow * T \vee coolant\_flow = 0\}$$

but this would be an *ad hoc* solution specific to this particular precondition and code.

In order to systematically account for, and deal with the difficulties that arise from undefined values, 3-valued logics were introduced [4, 8, 9, 10]. In this framework, the Boolean domain $\mathbb{B} = \{t, f\}$ is extended to $\mathbb{B}^{\perp} = \{t, f, \perp\}$, where $\perp$ designates the value "undefined", and the logical connectors such as OR, AND, and NOT are given semantic extensions with respect to $\mathbb{B}^{\perp}$. We would like to point out that semantic extensions of logical connectors can be done in a number of ways, e.g., Gries provides an asymmetric extension of OR, Jones provides a symmetrical one. In spite of the fact that undefined values have been accounted for in some sense, in the past, formal treatment of undefined values (resulting from the evaluation of partial functions) beyond propositional logic have been somewhat incomplete. One source of this incompleteness results from the fact that, in general, to describe a system in a natural manner, one may need several different extensions of each classical Boolean connector.

Indeed, in the example above, "$\vee$" in the precondition should be replaced by the nonstrict symmetric monotone extension of "$\vee$" over $\mathbb{B}^{\perp}$ (as that in [8]) which is designated as "$\vee_s$" in the following sections, whereas "||" in the code fragment should be interpreted as a monotone extension of "$\vee$" over $\mathbb{B}^{\perp}$ with a nonstrict right argument (as that in [4]) which is designated as "$\vee_r$" in the following sections. Note that $\vee_r$ is a precise semantics for the "short circuit" evaluation rule for "||" in C/C++ [12], whereas using "$\vee_r$" in lieu of "$\vee_s$" in the precondition is not natural since it would preclude treating the left argument of "OR" as nonstrict.

In the following sections we will provide several different (monotone) extensions of each classical Boolean connector.

1.2.2. Providing a Uniform Treatment of Partial Functions

1.2.2.1. From Partiality to Totality

Consider a one dimensional integer 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 a code fragment then f must be included in some form in the postcondition for the code fragment, e.g., $\{f(n)>0\}$. Since the software may include instructions such as $n := 101$, we have to replace the above formula by one which is meaningful for values of n beyond the domain of f. For example, we may want to consider the postcondition $\{1\le n\le 100 \wedge_s f(n)>0\}$ instead. Similarly to the preceding example, $\wedge_s$ is the nonstrict symmetric monotone extension of $\wedge$, defined later in the paper. (Thus if $n = 101$, the new form of the postcondition is false.) Now we may forget that f is partial and may arbitrarily extend it to a total function. This will not change the truth value of the precondition for any n. It can be found by ordinary theorem provers based on 2-valued logic.

This is the essence of the following idea of Gries [4]:

- 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 to 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 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?

In the following sections we will provide positive answers to the above questions.

1.2.2.2. From Totality to Partiality

An implementation of the above Gries idea is not yet sufficient for a uniform treatment of partial functions, since there are legitimate usages of total functions within software when only a part of the argument list is available. This is equivalent to regarding such total functions as partial on an extended domain. E.g., consider the "selection" function $(b\ ?\ x : y)$ from C/C++. It is obviously a total function when b,x, and y are defined. However, what is the meaning of (**true** ? 1 : 1/0) ? It is 1, even if the third argument is undefined i.e., not available. Thus, although, by itself, $(b\ ?\ x : y)$ is total, its usage does not conform to the classical Tarski's semantics, since the latter does not allow undefined arguments.

In the following sections we will provide a mechanism to account for such usages of total functions.

1.2.3. Extending the Proof Rules for Hoare Triples to Account for Partial Functions

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 [9], an excellent book on program correctness and derivation. It is suggested there (and in many other books and papers, e.g., [4, 9, 19], 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 connectors $\wedge$ and $\Rightarrow$ must be extended to cover undefined arguments, since $Q(x/E)$ may become undefined due to occurrences of partial functions in Q and/or E. This is discussed in previous sections;
- even if E does not have occurrences of partial functions, the formula may become undefined if P or Q contain partial functions. E.g., it is possible that $Def.E$ holds when $Q(x/E)$ is undefined.

In the following sections we will provide a formal definition of *Def* and will 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.

1.3. The Existing Research on Partial Functions

Many researchers worked in the area of partial functions and their applications in computing [10 and references there, 4, 13, 8, 2, 5].

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). Most of the researches, including Gries [4] and Jones [8], used 3-valued logics described in [10] and introduced various versions of explicit domains for partial functions. Note however, that Kleene's purpose was to elucidate partial functions in recursion theory, rather than to reason about software.

In order to provide a uniform treatment of partial functions, we introduce a new notion of nonstrict explicit domains that are substantially different from the ones previously considered.

2. NONSTRICT EXPLICIT DOMAINS AND CONSTRUCTION OF *DEF*.

2.1. Basic Definitions

We are working here with sorted partial algebras with explicit domains (see [17]). We will provide necessary definitions in an informal manner, see [17] for completely formal definitions. The expressions are defined inductively as follows:

- a variable $x{:}S$ or a constant $c{:}S$ are expressions of sort S;
- if $f{:}S_1\times...\times S_n \dashrightarrow S$ is a (partial) function and $t_1, ..., t_n$ are expressions then:
  - $f(t_1, ..., t_n)$ is an expression of sort S. If f is a Boolean-valued function, then $f(t_1, ..., t_n)$ is a Boolean expression;
- 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 in b becomes a bound occurrence in the above expressions. $\overset{ns}{\forall}$ and $\overset{ns}{\exists}$ are called the nonstrict quantifiers.

Following [4], we view each partial function $f{:}S_1\times...\times S_n \dashrightarrow S$ as such total function $\tilde{f}{:}S_1\times...\times S_n\to S$ that:

- $f(x_1, ..., x_n) = \tilde{f}(x_1, ..., x_n)$, whenever $(x_1, ..., x_n)\in Dom.f$
- $\tilde{f}(x_1, ..., x_n)$ is *unknown*, otherwise,

where $Dom.f$ is the domain of f in the usual sense. In order to formalize "*known*" and "*unknown*" we construct an expression transformer *Def* such that for every expression *exp*, *Def*(*exp*) is a Boolean expression such that if we replace every partial function f occurring in *exp* by its total extension $\tilde{f}$, thus producing an expression *exp'*, then

(A1) the value of *Def*(*exp'*) does not depend on how each extension $\tilde{f}$ is defined outside *Dom.f*.

(A2) if *Def*(*exp'*) = t then the value of *exp'* does not depend on how each extension $\tilde{f}$ is defined outside *Dom.f*.

Thus we can say that the value of an expression *exp* is *known* (or is *defined*) if *Def*(*exp*) = t and is unknown otherwise.

We construct *Def* by associating with each (partial) function $f{:}S_1\times...\times S_n \dashrightarrow S$ its nonstrict explicit domain $Edom.f{:}\mathbb{B}\times S_1\times ... \times\mathbb{B}\times S_n\to\mathbb{B}$ (see below) and defining *Def*(*exp*) as

- if $exp = f(t_1, ..., t_n)$, $f{:}S_1\times...\times S_n \dashrightarrow S$ is a function and $t_1, ..., t_n$ are expressions then:
  - $Def(exp) \triangleq Edom.f(Def.t_1, t_1, ..., Def.t_n, t_n)$;
- if b is a Boolean-valued expression, then
  - $Def(\forall x{:}S,\ b) \triangleq \forall x{:}S,\ Def.b$;
  - $Def(\exists x{:}S,\ b) \triangleq \forall x{:}S,\ Def.b$;
  - $Def(\overset{ns}{\forall} x{:}S,\ b) \triangleq (\forall x{:}S,\ Def(b)) \vee \exists x{:}S(Def.b \wedge_s \neg b)$;
  - $Def(\overset{ns}{\exists} x{:}S,\ b) \triangleq (\forall x{:}S,\ Def(b)) \vee \exists x{:}S(Def.b \wedge_s b)$,

In order to satisfy the conditions A1 and A2, we impose the following restrictions on the nonstrict explicit domains. For simplicity, let $f:S_1 \times S_2 \dashrightarrow S$. Then *Edom.f* satisfy the following:

- the standard set-theoretical domain may be computed as $Dom.f = \{(x, y) \mid Edom.f(\mathrm{t}, x, \mathrm{t}, y) = \mathrm{t}\}$;
- if $Edom.f(\mathrm{t}, x_0, \mathrm{f}, y) = \mathrm{t}$, 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 $Edom.f(\mathrm{f}, x, \mathrm{t}, y_0) = \mathrm{t}$, 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.

Informally, when considering $Edom.f(z, x, w, y)$, z means "x is defined" and w means "y is defined". For a full treatment of the explicit domains see [17]. We treat constants as functions of arity 0. If f is a known constant, *Edom.f* is a Boolean valued function of arity 0 whose single value is t. If f is *not* a known constant, *Edom.f* is a Boolean valued function of arity 0 whose single value is f. The examples of unknown constants are $\perp$ and program variables which are not initialized.

There may be distinct nonstrict explicit domains associated with identical functions. We are going to differentiate between such functions by assigning to them unique names. Finally, in order to use classical logic for computations, we will represent each partial function f by a pair $(\tilde{f}, Edom.f)$, where $\tilde{f}$ is a total extension of f. When thinking of $\tilde{f}$ as itself, $\tilde{f}(x, y)$ is known whenever both arguments are known. However, when we think of $\tilde{f}$ as a representation of f, $\tilde{f}(x, y)$ is known only if $Edom.f(\mathrm{t}, x, \mathrm{t}, y) = \mathrm{t}$.

2.2. Examples of Nonstrict Explicit Domains

- Boolean constants:
  - Edom.t = t;
  - Edom.f = t;
- The "undefined" element:
  - Edom.$\perp$ = f;
- Strict Boolean connectors and equality $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, and =. If $\star$ is of any of $\wedge$, $\vee$, $\Leftrightarrow$, $\Rightarrow$, and =, then:
  - Edom.$\star$(z, x, w, y) = z $\wedge$ w;
- Nonstrict Boolean connectors $\wedge_s$, $\vee_s$, $\Rightarrow_s$, $\wedge_r$, $\vee_r$, $\Rightarrow_r$, $\wedge_l$, $\vee_l$, $\Rightarrow_l$:
  - Edom.$\vee_s$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ x) $\vee$ (w $\wedge$ y);
  - Edom.$\vee_r$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ x);
  - Edom.$\vee_l$(z, x, w, y) = (z $\wedge$ w) $\vee$ (w $\wedge$ y);
  - Edom.$\wedge_s$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ $\neg$x) $\vee$ (w $\wedge$ $\neg$y);
  - Edom.$\wedge_r$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ $\neg$x);
  - Edom.$\wedge_l$(z, x, w, y) = (z $\wedge$ w) $\vee$ (w $\wedge$ $\neg$y);
  - Edom.$\Rightarrow_s$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ $\neg$x) $\vee$ (w $\wedge$ y);
  - Edom.$\Rightarrow_r$(z, x, w, y) = (z $\wedge$ w) $\vee$ (z $\wedge$ $\neg$x);
  - Edom.$\Rightarrow_l$(z, x, w, y) = (z $\wedge$ w) $\vee$ (w $\wedge$ y);

The idea for some of this explicit domain was obtained from the truth-table for the nonstrict monotone extensions of the logical connectors over the extended domain of Booleans given in [10, 8]. E.g., the truth-table for the symmetric monotone extension of OR over extended domain of Booleans is the following:

$\vee_s$	t	f	$\perp$
t	t	t	t
f	t	f	$\perp$
$\perp$	t	$\perp$	$\perp$

- Conditional function

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

- Edom.?$(z, b, v, x, w, y) \triangleq z \wedge (b \Rightarrow v) \wedge (\neg b \Rightarrow w)$.

2.3. The Five-Step Process for Evaluating Expressions with Partial Functions and/or Extended Logical Connectors

Let φ be a closed expression. Since φ is nonclassical (due to occurrences of partial functions and/or extended logical connectors), automated theorem provers cannot be applied to φ directly. We will show how to automate the process without ad hoc rewriting of φ.

Step 1. Transform φ into a formula *Def.*φ. Using the expression transformations facilities of a theorem prover (such as the ones available in OTTER [14]) or an automated term rewriting system, the transformation "$\varphi \rightarrow$ *Def.*φ" can be carried out automatically.

Step 2. Replace all the occurrence of nonclassical Boolean connectors in *Def.*φ by their classical counterparts (e.g., $\vee_s$ by $\vee$) and all the partial functions by their total extensions. We designate the result as *Classic(Def.*φ*)* since it falls within the classical two-valued first-order logic. This can be carried out automatically as well.

Step 3. Find the logical value of *Classic(Def.*φ*)* via a theorem prover. We will prove below that this value does not depend on how the extensions of the partial functions to total are carried out. If *Classic(Def.*φ*)* is false, then the original formula is undefined. In this case we **stop** here. Otherwise, if *Classic(Def.*φ*)* is true, proceed to step 4.

Step 4. Replace all the occurrence of nonclassical Boolean connectors in φ by their classical counterparts and all the partial functions by their total extensions. We designate the result as *Classic(*φ*)*.

Step 5. Find the logical value of *Classic(*φ*)* via a theorem prover. We will prove below that this value does not depend on how the extensions of the partial functions to total are carried out.

2.4. Application of the Five-Step Process to the Nuclear Reactor Example

2.4.1. The "Bottom" Example

Let's attempt to find the logical value of $\bot \vee_s \mathfrak{f}$. Recall that *Def*$(\bot) \triangleq \mathfrak{f}$ and *Def*$(\mathfrak{f}) \triangleq \mathfrak{t}$. Let us employ $\mathfrak{t}$ as an extension of $\bot$. We'll apply the 5-step procedure:

Step 1. *Def*$(\bot \vee_s \mathfrak{f}) \triangleq$ (*Def*$(\bot) \wedge$ *Def*$(\mathfrak{f})) \vee$ (*Def*$(\bot) \wedge \bot) \vee$ (*Def*$(\mathfrak{f}) \wedge \mathfrak{f}) \triangleq (\mathfrak{f} \wedge \mathfrak{t}) \vee (\mathfrak{f} \wedge \bot) \vee (\mathfrak{t} \wedge \mathfrak{f})$.
Step 2. *Classic(Def*$(\bot \vee_s \mathfrak{f})) \triangleq (\mathfrak{f} \wedge \mathfrak{t}) \vee (\mathfrak{f} \wedge \mathfrak{t}) \vee (\mathfrak{t} \wedge \mathfrak{f})$.
Step 3. $(\mathfrak{f} \wedge \mathfrak{t}) \vee (\mathfrak{f} \wedge \mathfrak{t}) \vee (\mathfrak{t} \wedge \mathfrak{f}) \triangleq \mathfrak{f}$.

Thus the formula is undefined and we can't find its logical value. Note that if we would transpose Step 2 and Step 1, i.e., substitute the extension of $\bot$ directly into the formula $\bot \vee_s \mathfrak{f}$, we would have a defined formula. This shows that the 5-step procedure is not commutative. Finally, it's easy to check that result of Step 3 would be the same if we would employ $\mathfrak{f}$ as an extension of $\bot$.

2.4.2. The Nuclear Reactor Example

Let's consider as an example the precondition

$$exp \triangleq heat/coolant\_flow \geq T \vee coolant\_flow = 0$$

from the nuclear reactor example. We consider *heat, coolant_flow, T, coolant_flow,* and 0 to be known and therefore their explicit domains are all equal to $\mathfrak{t}$. We will treat "$\geq$", "$\vee$", and "$=$" as Boolean-valued binary functions. Thus the above expression has occurrences of four binary functions, namely "/", "$\geq$", "$\vee$", and "$=$". All these functions are strict in the sense that their values are known only if all the arguments are known. The division "/" is the only partial function out of the four. Let us extend "/" by a function $Div(x, y)$ such that $Div(x, 0) = 0$ for each$x$. The explicit domains are the following:

- *Edom./*$(z, x, w, y) \triangleq z \wedge w \wedge (y \neq 0)$;
- *Edom.*$\star(z, x, w, y) \triangleq z \wedge w$,

where "$\star$" is any of "$\geq$", "$\vee$", and "$=$". We emphasize that the above form for *Edom.*$\vee$ was chosen because the classical understanding of OR is that both disjuncts are known.

Now, let's assume *coolant_flow* $\triangleq 0$ and compute *Def(exp)* using the above definitions:

Step 1. $Def(exp) \triangleq Def(heat/0 \geq T) \wedge Def(0 = 0) \triangleq Def(heat/0) \wedge Def(T) \wedge Def(0) \wedge Def(0) \triangleq Def(heat) \wedge Def(0) \wedge 0 \neq 0 \wedge \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{t} \triangleq \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{f} \wedge \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{t}$.

Step 2. $Classic(Def.exp) \triangleq Def(exp) \triangleq \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{f} \wedge \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{t}$. Note that in this case there are no occurrences of partial functions in *Def(exp)*.

Step 3. $\mathrm{t} \wedge \mathrm{t} \wedge \mathrm{f} \wedge \mathrm{t} \wedge \mathrm{t} \wedge \mathrm{t} \triangleq \mathrm{f}$.

This means that $heat/coolant\_flow \geq T \vee coolant\_flow = 0$ is undefined when $coolant\_flow = 0$. This is not satisfactory, since we wish to be able always to check whether this particular precondition is true or false. Therefore, we will replace the classical explicit domain for OR by such another one which would enable us to compute x OR y whenever one disjunct is known and another is unknown. Since we wish to preserve unique names associated with explicit domains, we'll rename new connector as $\vee_s$. The explicit domain for this new connector is as follows:

$$Edom.\vee_s(z, x, w, y) \triangleq (z \wedge w) \vee (z \wedge x) \vee (w \wedge y).$$

Now let $exp' \triangleq heat/coolant\_flow \geq T \vee coolant\_flow = 0$. Let's apply the 5-step procedure to *exp'* when $coolant\_flow \triangleq 0$:

Step 1. $Def(exp') \triangleq (Def(heat/0 \geq T) \wedge Def(0 = 0)) \vee (Def(heat/0 \geq T) \wedge heat/0 \geq T) \vee (Def(0 = 0) \wedge 0 = 0) \triangleq \mathrm{f} \vee (\mathrm{f} \wedge heat/0 \geq T) \vee (\mathrm{t} \wedge \mathrm{t}) \triangleq (\mathrm{f} \wedge heat/0 \geq T) \vee \mathrm{t}$.

Step 2. $Classic(Def.exp') \triangleq (\mathrm{f} \wedge Div(heat, 0) \geq T) \vee \mathrm{t} \triangleq (\mathrm{f} \wedge Div(heat, 0) \geq T) \vee \mathrm{t}$.

Step 3. $(\mathrm{f} \wedge Div(heat, 0) \geq T) \vee \mathrm{t} \triangleq (\mathrm{f} \wedge 0 \geq T) \vee \mathrm{t} \triangleq \mathrm{t}$. Note that the result does not depend either on the value of $Div(heat, 0)$ or on the value of T.

Step 4. $Classic(exp') \triangleq Div(heat, 0) \geq T \vee 0 = 0$.

Step 5. $Div(heat, 0) \geq T \vee 0 = 0 \triangleq \mathrm{t}$.

That's exactly what the intuitive meaning of this formula should give.

2.5. Theorems Justifying the Approach

Let t be a closed expression.

Theorem 1. *Classic(Def.t)* is a closed expression and its Boolean value does not depend on the choice of total extensions of partial functions occurring in t. □

Theorem 2. If $Classic(Def.t) = \mathrm{t}$ then the value of *Classic(t)* does not depend on the choice of total extensions of partial functions occurring in t. □

The proofs of these theorems can be found in [17].

3. AUTOMATING CORRECTNESS PROOFS OF SOFTWARE WITH PARTIAL FUNCTIONS

3.1. Application of the Five-Step Process to the Correctness Proofs

In this paper we focus on a subset of the Dijkstra language which includes the following components: assignments, conditionals, and special loops which we call simple verifiable loops described at the end of the paper. Suppose c is a code fragment in this language. For every Hoare triple $\{P\}c\{Q\}$ which may contain partial functions and/or extended logical connectors, we construct the following formula:

$$(Def.P \wedge_s P) \Rightarrow_s wpp(c, Q),$$

where *wpp(c, Q)* denotes the formula representing the "weakest precondition in the presence of partial functions". The formula *wpp(c, Q)* is constructed using our rules provided at the end of the paper. Our rules extend the Dijkstra weakest precondition rules. Let us designate $(Def.P \wedge_s P) \Rightarrow_s wpp(c, Q)$ as φ. Since φ is nonclassical (due to occurrences of partial functions and/or extended logical connectors), automated theorem provers cannot be applied to φ directly. In order to find whether φ is correct, we'll apply the 5-step process above.

Theorem 3. The following is true:

- If *Classic(Def.φ)* is true, then the Hoare triple *{P}c{Q}* is defined in the sense that if the code fragment *c* begins its execution at any state s_0 such that $P(s_0)$ is defined and holds, then

  - during the execution of *c* there will be no attempts to compute the value of any partial function outside its domain.
  - upon termination, *c* produces a state s_1 such that $Q(s_1)$ is defined.

- If *Classic(Def.φ)* is false, then the Hoare triple *{P}c{Q}* is undefined in the sense that at least for one state s_0 such that $P(s_0)$ is defined and holds, at least one of the two condition above will be violated.

Proof. Induction on the length of *c*.

□

Theorem 4. If *Classic(Def.φ)* is true then the following is true:

- If *Classic(φ)* is true, then the Hoare triple *{P}c{Q}* is correct in the sense that if the code fragment *c* begins its execution at any state s_0 such that $P(s_0)$ is defined and holds, then

  - during the execution of *c* there will be no attempts to compute the value of any partial function outside its domain.
  - *c* terminates
  - upon termination, *c* produces a state s_1 such that $Q(s_1)$ is defined and holds.

- If *Classic(φ)* is false, then the Hoare triple *{P}c{Q}* is incorrect in the sense that at least for one state s_0 such that $P(s_0)$ holds, at least one of the three condition above will be violated.

Proof. Induction on the length of *c*.

□

3.2. Semantics of Programs with Partial Operations

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. We identify the states with first order structures which have signatures including the signature of the data structure viewed as an algebra. Although there are many descriptions of formal program semantics, see [11], the most convenient for us is the "evolving algebras" semantics developed by Y. Gurevich, see [5]. In [17] we modified the original evolving algebras to accommodate our explicit domains, thus obtaining evolving sorted partial algebras with explicit domains (ESPED-algebras). We also defined there (via ESPED-algebras) a semantics of programs in a subset of the Dijkstra language. The semantics of programs is necessary to prove the theorems 3 and 4 about Hoare triples with partial functions (see the previous subsection). These theorems show the soundness of the proof rules for the Hoare triples given in the following section.

Here we'll only present an informal operational semantics for the language.

Instructions	Behavior during Execution
Skip ***skip***	Step 1. Do nothing; Step 2. Terminate.
Composition /* *F* and *G* are algorithms */ *F*; *G*	Step 1. Execute *F*; Step 2. Execute *G*; Step 3. Terminate.
Simple Assignment /* *x* is a variable and *E* 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 ṫ, and leaving the values of all other variables unchanged; Step 3. Terminate.

Strict Indexed Assignment /* let $f:S_1, ..., S_n \rightarrow S$ 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 f 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 $Edom.f(t, t_1, ..., t, t_n)$ by t and leaving the values of all other variables unchanged; Step 3. Terminate.
Simple IF /* γ is a Boolean expression and F and G are algorithms. */ **if** $\gamma \rightarrow F$ $\square \neg\gamma \rightarrow G$ **fi**	Step 1. Evaluate $Def(\gamma)$ in the initial program state. If $Def(\gamma) = f$ then crash. Otherwise go to the next step; Step 2. If γ it evaluates as t, execute F. Otherwise execute 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 F is an algorithm. It is established that φ is an invariant of F and that E is a bound function. */ **do** $\gamma \rightarrow$ **invariant** φ **bound function** E F **od**	/* The following must be proved beforehand: • $\{\varphi \wedge_s \gamma\}\ F\ \{\varphi\} \wedge_s (Def(\varphi) \wedge_s \varphi \Rightarrow_s Def(\gamma))$, i.e., φ is an invariant of the loop; • $(\varphi \Rightarrow_s E \geq 0) \wedge_s \{E=X\}\ F\ \{E<X\}$, where X is an integer program variable not occurring in F. Thus E is a bound function of the loop. */ Step 1. Evaluate $Def(\varphi)$ in the initial program state. If $Def(\varphi) = f$, then crash. Otherwise go to step 2; Step 2. Evaluate the loop guard γ. If γ evaluates as f, then terminate. Otherwise go to step 3; Step 3. Execute the loop body F. When and if F terminates, go to step 2.
Pseudocode Instruction /* SV is a list of program variables called "specification variables", φ, ψ are logical assertions */ $\|[SV, \varphi, \psi]\|$	Step 1. Evaluate $Def(\varphi)$ in the initial program state. If $Def(\varphi) = f$, or if $Def(\varphi) = t$ and $\varphi = f$ 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.

3.3. Extending the Dijkstra-Gries Program Correctness Rules

We will extend the Dijkstra weakest precondition (wp) expression transformer to programs with partial operations. We'll denote the new transformer as 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 $F; G$	• $wpp(F; wpp(G, Q))$
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** $\gamma \rightarrow F$ $\square \neg\gamma \rightarrow G$ **fi**	• $Def(\gamma) \wedge_s (\gamma \Rightarrow wpp(F, Q)) \wedge_s (\neg\gamma \Rightarrow wpp(G, Q))$

Simple Verifiable Loop **do** $\gamma \rightarrow$ **invariant** φ **bound function** E F **od**	• φ • $\uparrow$	if $Def(\varphi) \wedge_s \varphi \wedge_s \neg\gamma \Rightarrow Def(Q) \wedge_s Q$; otherwise.
Pseudocode Instruction $\|[SV, \varphi, \psi]\|$	• φ • $\uparrow$	if $Def(\varphi) \wedge_s \varphi \Rightarrow Def(Q) \wedge_s Q$; otherwise.

4. SUMMARY

When using a computer to evaluate expressions containing partial functions care must be taken to avoid the actual evaluation of undefined values. For example, consider the evaluation of the expression $2/0 = 1 \wedge 1 = 2$. In order to evaluate it in a three-valued logic, we'll rewrite it as $2/0 = 1 \wedge_s 1 = 2$ where $\wedge_s$ corresponds to conjunction extended to symmetric and monotone function over extended Booleans, thus "undefined" $\wedge_s$ **false** yields **false**. With respect to this three-valued logic, the evaluation of the above expression will produce the value **false.** However, even though the expression $2/0 = 1 \wedge_s 1 = 2$ has a meaning according to this three-valued logic, a computer will encounter difficulties if it is asked to evaluate the subexpression 2/0. The evaluation of 2/0 demonstrates an instance of a class of problems (i.e., evaluation of undefined values) that are encountered in attempts to use the automated realization of three-valued logics to model computations with partial functions.

In the framework that we have presented in this paper, we define when a compound expression or formula with partial functions can be computed and when it can not be computed. We do this by providing a rigorous definition of the domains of applicability of expressions and formulas. The key elements of this definition are predicates *Edom* and *Def*, together with an extension of partial functions to total functions. These provide a mechanism for rewriting a formula φ to φ'. The objective of this rewriting process is to produce a formula φ' having the desirable property that it consists exclusively of total functions, while the truth-value of the formula does not depend on which particular total extensions of partial functions are selected.

For example, let us consider the evaluation of the expression $2/0 = 1 \wedge_s 1 = 2$ in our framework. The predicates *Edom* and *Def* describe for what inputs the /, =, and $\wedge_s$ are defined. Using this domain information it can then be determined that $Def(2/0 = 1 \wedge_s 1=2)$ holds. From this we conclude that all partial functions can be extended (by arbitrary values) to total functions and all extended logical connectors can be replaced by their classical analogs, i.e., $\wedge_s$ can be replaced by $\wedge$. Now, for the sake of interest, suppose / has been extended so that $2/0 = 1$. For such an extension the evaluation of $2/0 = 1 \wedge_s 1 = 2$ will produce $1 = 1 \wedge 1 = 2$ as an intermediate result. In turn this will yield **true** $\wedge$ **false** which evaluates to **false**. Other extensions of /, such as $2/0 = 0$ will produce the same result. Not surprisingly, this is the same value that resulted from the evaluation of $2/0 = 1 \wedge_s 1 = 2$ by means of the original three-valued propositional logic. Note however that this expression can be evaluated within the three-valued propositional logic only if we supply the information that $(2/0 = 1)$ is "undefined". An attempt to find this by actually computing $2/0 = 1$ would cause the same difficulties that we have been trying to avoid.

Finally, we have done the following

- introduced a new notion of *nonstrict explicit domains* of partial functions represented within the classical first order predicate logic. This allowed us to model each partial function f as a pair consisting of the nonstrict explicit domain of f and an arbitrary total extension of f;
- provided rigorous and uniform definitions of the set-theoretical domains of expressions including partial functions. This domains are also represented within the classical first order predicate logic. They are constructed by utilizing the above pairs modeling the partial functions in the expressions;
- developed models of functions with argument lists of variable length. This is done via our *nonstrict explicit domains*. Thus such languages as C/C++ would be able to enter in the realm of program correctness proofs.
- provided a process for verification of Hoare triples containing partial functions which permits us to use existing theorem provers which were not designed to accommodate partial functions.

ACKNOWLEDGMENTS

Alex and Vlad 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. Alex and Vlad are grateful to David Gries for teaching them the program derivation.

BIBLIOGRAPHY

1. Apt, K.R., Olderog, E. R., *Verification of Sequential and Concurrent Programs*, Springer-Verlag, 1991.
2. Breu, R., Algebraic Specification Techniques in Object Oriented Programming Environments, Springer-Verlag, 1991.
3. Dijkstra, E.W., Scholten, C.S., *Predicate Calculus and Program Semantics*, Springer-Verlag, 1990.
4. Gries, D., *The Science of Programming*, Springer-Verlag, 1981.
5. Gurevich, Y., Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, pp. 7-36, Oxford University Press, 1995.
6. Guttag, J.V., Horning, J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
7. 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.
8. Jones, C.B., *Systematic Software Development using VDM*, Prentice-Hall International 1990.
9. Kaldewaij, A., *Programming: The Derivation of Algorithms*. Prentice Hall, 1990.
10. Kleene, S., *Introduction to Metamathematics*, D. Van Nostrand Co., 1952
11. Loeckx, J., Sieber, K., *The Foundations of Program Verification*, John Wiley & Sons, 1987.
12. Pohl, I., *Object-Oriented Programming Using C++*, Benjamin/Cummings, 1993.
13. Reichel, H., *Initial computability, Algebraic Specifications, and Partial Algebras*, Claredon Press, Oxford, 1987
14. McCune, W.W., OTTER 3.0 Users Guide. Technical Report ANL-94/6, Argonne National Laboratories, 1994.
15. Tucker, J.V., Zucker, J.I., *Program Correctness over Abstract Data Types with Error-State Semantics*, North-Holland, 1988.
16. Winter, V., *Proving the Correctness of Program Transformations*. Ph.D. Thesis, 1994.
17. Yakhnis, A., Yakhnis, V., Semantics and Correctness Proofs for Programs with Partial Functions, Sandia National Labs Technical Report No. 96-1123, 1996.
18. Yakhnis, A., Yakhnis, V., Reliable Software Systems via Chains of Object Models with Provably Correct Behavior, submitted to The Seventh International Symposium on Software Reliability Engineering, White Plains, NY, October 30 - November 2, 1996, also available as Sandia National Labs Technical Report No. 96-1192.
19. Yakhnis, V., Farrell, J., Shultz, S. (1994) Deriving Programs Using Generic Algorithms, *IBM Systems Journal*, vol. 33, no. 1, pp. 158-181, 1994.