# Formal Treatment of a Family of Fixed-Point Problems on Graphs by CafeOBJ

Tetsuo Tamai
Interfaculty Initiative in Information Studies
University of Tokyo
3-8-1 Komaba, Meguro-ku
Tokyo 153, Japan

## Abstract

*A family of well known problems on graphs including the shortest path problem and the data flow analysis problem can be uniformly formulated as a fixed-point problem on graphs. We specify this problem and its solution algorithm in a highly abstract manner, fully exploiting the parameterized module construct of CafeOBJ, an algebraic specification language. The objective of our research is to explore effectiveness of formal methods applying them not just to safety critical specific programs but to general problems covering wide range of applications.*

## 1. Introduction

There are claims that formal methods are already at the level of practical use and are actually widely used [4, 11]. In spite of those claims, cases often cited as model examples of successfully applying formal methods are limited in number. A typical example is the IBM's CICS re-specification project using Z [7], which has been referred by almost all survey papers on formal methods.

Typical targets of formal methods have been hardware logic, communication protocols and safety critical software. If applications are limited within those areas, it would be hard to appeal the usefulness of formal methods to wider software engineers and managers. Thus, efforts have been made to apply them to "real" middle or large scale system developments. One of the champions in this category is the CICS project and as the result so much citations.

However, the gap between the group of hardware, protocols and safety critical software and complex and large-scale real software systems is wide. There should be some area in the middle between these two that fits well to the formal description technique and also has a practical significance.

B. Meyer argued that the most appropriate targets of formal methods are reusable components [5]. By the same token, we claim that problem frameworks and their solution algorithms that can be applied to multiple domains and can be shared by wide-range application areas are another good target of formal methods. The descriptions of general problem frameworks and solutions should be highly abstract and had better be given in a formal language. Then, they can be reused at the high abstraction level and if their reliability is assured through some formal means, it will greatly improve the productivity and reliability of programs customized for specific domains.

In the following, we focus on a class of fixed point problems on graphs that can be uniformly described at an abstract level and can be applied to various areas, including the shortest path problem, data flow analysis problems, network reliability problems, AI search problems and integer programming problems [10]. The formulation and solutions of this problem can be directly applied to the program analysis and verification techniques familiar to the formal method community, i.e. model checking [3] and program dependency graph analysis [8].

The treatment of this problem in the author's paper [10] was informal. Yamamoto et al. gave a formal description of the problem using HOL and proved the correctness of its iterative type solution algorithm [13]. In this paper, we treat the same problem in CafeOBJ, an algebraic specification language.

## 2. Algebraic Specification

There can be a wide variety of ways for writing a formal specification of some specific problem. Firstly, we can choose a basic formal method and a language based on that method. Then, we should decide a fundamental structure of the specification and its description style. In our approach, we chose the albegraic specification approach, taking CafeOBJ as a specification language and fully exploited its modularaization constructs including parameterised modules and views.

## 2.1. CafeOBJ

CafeOBJ is one of the youngest descendants in the OBJ langauge family [2]. It is a multi-paradigm algebraic specification language, combining many sorted algebra, order sorted algebra, hidden sorted algebra and rewriting logic. Another activity of designing and implementing an algebraic specification language is seen in Europe, i.e. CASL by the CoFI group [1]. The reason we chose CafeOBJ is simply because the author has been a member of the CafeOBJ environment development project. In this paper, we only use many and order sorted algebra, the core of CafeOBJ.

A module in CafeOBJ encapsulates definitions of a sort or a set of sorts. A module consists of three parts: the **imports** block for references of other modules, the **signature** block for defining signature of operations belonging to the sorts to be specified and the **axioms** block for giving semantics.

## 2.2. Tight and Loose Modules

There are two kinds of modules, tight modules and loose modules. A tight module denotes a unique model corresponding to the initial algebra, whereas a loose module denotes a class of models without unique denotations. The former can be used for defining abstract data types in the familiar computer programming sense, eg. integer, stack, queue, etc., while the latter can be regarded as defining algebraic theories like group, ring, graph, etc. In CafeOBJ, the module declaration keyword `module` can be suffixed by ! to indicate the module has tight denotation and by * to indicate loose denotation.

## 2.3. Parameters and Views

A paremeterised module in CafeOBJ is a construct that provides a mechanism for passing other modules to the definition of a module. Instantiation of parameterised modules are done by replacing parameters with modules that satisfy certain syntactic and semantic constraints. For example, *list* can be defined as a sort in a parameterised module. A list of integers can be obtained by instantiating the parameter with a module defining the integer.

A view specifies a way to bind actual parameters to formal parameters. A view is a morphism from a module to another module. A view $V$ from a module $T$ to a module $M$ maps sorts and operators in $T$ to corresponding sorts and operators in $M$ and the axioms of $T$ must hold in $M$ after the substitution of corresponding operators. For example, suppose $T$ is a (loose) module specifying the partial order and $M$ is a (tight) module specifying the natural number, a view from $T$ to $M$ would define a mapping from the partial order relation to the order of the natural number in the usual sense.

# 3. Problem Description

The problem we deal with is a fixed point problem on graphs. We can formalize the problem and describe a generic solution algorithm at a highly abstract level using mathematical concepts of lattice, function space on lattice and graphs. The general formulation can be interpreted and customized in specific domains so that the graph reachability problem, the shortest path problem, the data flow problems, the network reliability problem among others will be generated. This process of interpretation and customization is done by means of assigning concrete algebraic sets to the lattice and the functions on the lattice. Also, by instantiating the abstract graph with a concrete graph with specific topology and attributes labeling vertices and edges, we obtain a concrete problem to be solved.

Thus, our strategy of formulating the problem in CafeOBJ is as follows.

1. For the abstract description of the problem in terms of lattice, functions on lattice and graphs, we use loose modules.

2. For the interpretation of the problem in some specific domain, we write tight modules and specify views from the loose modules defined above to the tight modules in the concrete problem domain.

3. To specify a concrete problem, we describe a concrete graph with specific topology and attributes as a module that imports the abstract graph module and extends it.

## 3.1. Formulation

First, we define the concept of lattice.

Let $L$ be a lattice, that is $L$ has two binary operations, join $\vee$ and meet $\wedge$, that have properties described below, and is closed under these operations.

1. Both join and meet operations are commutative and associative.

$$x \wedge y = y \wedge x, \quad x \vee y = y \vee x,$$

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z, \quad x \vee (y \vee z) = (x \vee y) \vee z.$$

2. The absorption law,

$$x \vee (x \wedge y) = x \wedge (x \vee y) = x.$$

holds. This law implies the idempotent law,

$$x \wedge x = x, \quad x \vee x = x.$$

A partial order relation $\leq$ can be defined by:

$$x \leq y \ \leftrightarrow \ x \wedge y = x.$$

Lattice is a typical algebraic concept, which can be defined most conveniently using a CafeOBJ theory type module with loose denotation.

```
module* LATTICE {
  [ Elt ]
  signature {
    op _v_ : Elt Elt -> Elt
                     {assoc comm}
    op _^_ : Elt Elt -> Elt
                     {assoc comm}
    op _<=_ : Elt Elt -> Bool
  }
  axioms {
    vars x y : Elt
    eq x v (x ^ y) = x .
    eq x ^ (x v y) = x .
    eq x <= y = x ^ y == x .
  }
}
```

We also assume an extra property to our lattice.

3. Existence of the greatest element:
   There is the greatest element 1 in $L$, i.e. for all $x \in L$,

$$1 \wedge x = x.$$

This can be specified by extending the module LATTICE.

```
module* LATTICE-T {
  protecting(LATTICE)
  signature {
    op top : -> Elt
  }
  axioms {
    var x : Elt
    eq top ^ x = x .
  }
}
```

Let $F$ be a set of functions $L \to L$. We define two binary operations on $F$, composition $\cdot$ and meet $\wedge$ as follows.

Let $f, f_1, f_2 \in F$,

$$f = f_1 \cdot f_2 \ \leftrightarrow \ \forall x \in L.\, f(x) = f_1(f_2(x)),$$

$$f = f_1 \wedge f_2 \ \leftrightarrow \ \forall x \in L.\, f(x) = f_1(x) \wedge f_2(x).$$

Composition $\cdot$ is associative, whereas meet $\wedge$ is commutative and associative. Also, the following distributive law holds between $\cdot$ and $\wedge$.

$$(f_1 \wedge f_2) \cdot f = f_1 \cdot f \wedge f_2 \cdot f,$$

$$f \cdot (f_1 \wedge f_2) = f \cdot f_1 \wedge f \cdot f_2.$$

We assume that $F$ has the following properties.

1. It is closed under the operations $\cdot$ and $\wedge$.
2. A unit element $\iota$ for the operation $\cdot$ and a unit element $\tau$ for the operation $\wedge$ exist in $F$. $\iota$ is an identity function and $\tau$ is a constant function that always returns 1, i.e. for all $x \in L$,

$$\iota(x) = x, \quad \tau(x) = 1.$$

3. Any $f \in F$ is monotonous in the following sense:

$$\forall x, y \in L.\, (x \leq y \to f(x) \leq f(y)).$$

The corresponding CafeOBJ code is as follows. Here, the CafeOBJ construct of parameterized module is used.

```
module* TRANS[L :: LATTICE-T] {
  [ Trans ]
  signature {
    op _*_ : Trans Trans -> Trans
                               {assoc}
    op _^_ : Trans Trans -> Trans
                          {assoc comm}
    op _<_> : Trans Elt -> Elt
    ops iota tau : -> Trans
  }
  axioms {
    vars f f1 f2 : Trans
    vars x y : Elt
    eq (f1 * f2) < x >
        = f1 < (f2 < x >) > .
    eq (f1 ^ f2) < x >
        = (f1 < x >) ^ (f2 < x >) .
    eq iota < x > = x .
    eq tau < x > = top .
    ceq (f < x >) <= (f < y >)
         = true if x <= y .
  }
}
```

Let $G = (V, E)$ be a directed graph with a vertex set $V$ and an edge set $E$. An edge $e$ corresponds to an ordered vertex pair $(v, w)$; the vertex $v$ is called a starting vertex and $w$ an ending vertex of $e$. Mapping from an edge to its starting vertex is represented by $h$ (i.e. $h(e) = v$), and mapping to its ending vertex is represented by $t$ (i.e. $t(e) = w$).

We define a theory module GRAPH at a very abstract level.

```
module* GRAPH {
  [ Vertex Edge ]
  signature {
    ops head tail : Edge -> Vertex
  }
}
```

We define an incoming edge set to a vertex $v$, in$(v)$, and an outgoing edge set from $v$, out$(v)$, by the followings.

$$\text{in}(v) = \{e \in E | t(e) = v\}, \quad \text{out}(v) = \{e \in E | h(e) = v\}.$$

We also assume the existence of a specific vertex *source*. For the sake of brevity, we omit the corresponding description in CafeOBJ but a module GRAPHEX is defined extending the module GRAPH with the properties above.

Our problem can be described as follows. Each edge $e$ in $E$ is labeled by an element $f_e$ of $F$. Each vertex $v$ in $V$ is labeled by a variable $x_v$ over $L$ and it satisfies the fundamental equation:

$$\begin{aligned} x_s &= b_s, \\ x_v &= \bigwedge_{e \in \text{in}(v)} f_e(x_{h(e)}) \quad (v \neq s). \end{aligned} \tag{1}$$

Here, the vertex $s$ is the source and is given a constant initial value. The operation $\wedge$ is extended over a set of $L$ elements using its commutativity and associativity. The corresponding representation of the equations (1) is as follows.

```
module! MEETOVERLIST[L :: LATTICE-T] {
  protecting(LIST[X <= view to L
                   {sort Elt -> Elt}]
             * { sort List -> Llist })
  signature {
    op meet : Llist -> Elt
  }
  axioms {
    var e : Elt
    var el : Llist
    eq meet(nil) = top .
    eq meet( e el ) = e ^ meet(el) .
  }
}
module! FIXEDPOINT[G :: GRAPHEX,
                   L :: LATTICE-T] {
  protecting (TRANS[L])
  protecting (MEETOVERLIST[L])
  signature {
    op fe : Edge -> Trans
    op xv : Vertex -> Elt
    op mapfe : Elist -> Llist
    op bs : -> Elt
  }
  axioms {
    var v : Vertex
    var e : Edge
    var el : Elist
    eq mapfe(nil) = nil .
    eq mapfe( e el )
        = (fe(e) < xv(head(e)) >)
           mapfe(el) .
    eq xv(source) = bs .
    eq xv(v) = meet(mapfe(inedge(v))) .
  }
}
```

Figure 1 shows the relations between the modules so far, where a vertical arrow denotes an import relation (the above is imported by the below module) and a horizontal arrow with a label denotes an import through a parameter named by the label.

```
LATTICE
   |
   V
LATTICE-T --L--> TRANS ----
          |                |
          --L--> MEETOVER  |        GRAPH
          |             |  |          |
          |             V  V          V
          --L--> FIXEDPOINT <--G-- GRAPHX
```

**Figure 1. Relations between modules around FIXEDPOINT**

## 3.2. Problem Example

A number of specific problems can be treated under the formulation introduced above as shown in the Table 1.

Here, we just give an example of the shortest path problem.

**Shortest path**
Let $L$ be a non-negative integer set with a lattice structure introduced by the ordinary order, thus the operator $\wedge$ corresponds to $min$ (, dually, the operator $\vee$ corresponds to $max$ but we do not use it). We add an element $\infty$ to $L$ as the greatest element. In CafeOBJ, we first define a module of natural number with min and max operations and then give a "view" from the module of lattice to this module as follows.

**Table 1. Specialization to various problem domains**

| problems | lattice | functions |
|---|---|---|
| reachability | (0, 1) Boolean lattice<br>0: reachable<br>1: unreachable | identity function |
| shortest path | non-negative integer $+\{\infty\}$<br>$\wedge$: min | $f_e(x) = x + d_e$ |
| network reliability | polynomials over stochastic variables $P_i$<br>$\wedge$: product without variable duplication<br>$\vee$: $p \vee q \equiv p + q - (p \wedge q)$ | $f_e(x) = x \wedge p_e$ |
| finite automaton | set lattice of strings | $f_e(x) = x \cdot a_e$<br>string concatenation |
| data flow analysis | set lattice of data property sets | $f_e(x) = x \cap K_e \cup G_e$ |

```
module! NAT-MAXMIN {
  protecting(NAT)
  [ Nat < Nat+ ]
  signature {
    op inf : -> Nat+
    ops min max : Nat+ Nat+ -> Nat+
                    {assoc comm}
    op _<=_ : Nat+ Nat+ -> Bool
    op _+_ : Nat+ Nat+ -> Nat+
                    {assoc comm}
  }
  axioms {
    vars x y : Nat+
    eq x <= inf = true .
    eq x + inf = inf .
    ceq min(x,y) = x if x <= y .
    ceq min(x,y) = y if y <= x .
    ceq max(x,y) = x if y <= x .
    ceq max(x,y) = y if x <= y .
  }
}

view LATTICEtoNAT from LATTICE-T
                  to NAT-MAXMIN {
  sort Elt -> Nat+,
  op   _^_ -> min,
  op   _v_ -> max,
  op   top -> inf,
  op   _<=_  -> _<=_
}
```

A concrete example of a graph corresponding to Fig.2 can be described as follows.

```
module! GRAPH-A {
  [ Vertex Edge ]
  signature {
    ops v1 v2 v3 v4 v5 v6 : -> Vertex
```

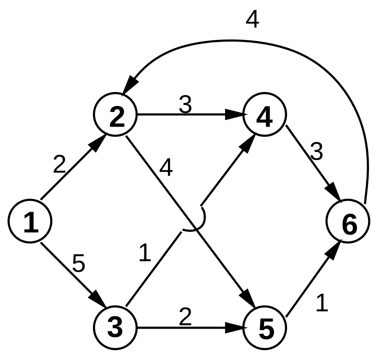

**Figure 2. An Example Graph for the Shortest Path Problem**

```
    ops e1 e2 e3 e4 e5 e6 e7 e8 e9 :
        -> Edge
    ops head tail : Edge -> Vertex
  }
  axioms {
    eq head(e1) = v1 . eq tail(e1) = v2 .
    eq head(e2) = v1 . eq tail(e2) = v3 .
  **> ... (rest omitted)
  }
}

module! GRAPH-AX {
   imports {
     protecting(GRAPH-A)
     protecting(LIST[X <= view to
         GRAPH-A {sort Elt -> Vertex}]
         * { sort List -> Vlist,
             op nil -> nil })
     protecting(LIST[X <= view to
         GRAPH-A {sort Elt -> Edge}]
         * { sort List -> Elist,
             op nil -> nil })
   }
   signature {
     op source : -> Vertex
     op inedge : Vertex -> Elist
     op outedge : Vertex -> Elist
   }
   axioms {
     eq source = v1 .
     eq inedge(v1) = nil .
     eq inedge(v2) = e1 e9 nil .
  **> ... (rest omitted)
     eq outedge(v1) = e1 e2 nil .
     eq outedge(v2) = e3 e4 nil .
  **> ... (rest omitted)
   }
}
```

With a distance $d_e$ associated with each edge $e$, a function is defined as:

$$f_e(x) = x + d_e.$$

When we solve the problem (1) with $b_s = 0$, the solution $x_v$ gives the shortest distance from the initial vertex $s$ to $v$.

By giving a distance to each edge of the graph, the problem can be described as follows.

```
module! SHORTESTP {
  protecting (FIXEDPOINT[G <= GRAPH-AX,
                         L <= LATTICEtoNAT])
  signature {
    op dist : Edge -> Nat+
  }
  axioms {
    var e : Edge
    var x : Nat+
    eq dist(e1) = 2 .
    eq dist(e2) = 5 .
  **> ... (rest omitted)

    eq fe(e) < x > = x + dist(e) .
  }
}
```

Thus, the module SHORPTESTP is an instance of FIXEDPOINT with tight denotation, where the parameter G is instantiated by a tight module GRAPH-AX and the parameter L is instantiated by a tight module NAT-MAXMIN.

## 4. Iterative Solution Method

### 4.1. Generic Algorithm

There can be two basic types of algorithms, the iteration type and the elimination type, just like the solution methods for linear equation systems. In this paper, we focus on the iterative method.

A generic iterative algorithm can be described informally as follows.

**[Generic Iterative Algorithm]**

**Input** Graph $G = (V, E)$, an initial vertex $s \in V$, lattice $L$, function space $F$ and a set of functions $f_e \in F$ are given, whose definitions are as stated before. A constant $b_s (\neq 1)$ is given to the initial vertex $s$ and the equations of the form (1) are considered. We further assume that for any $x \in L - \{1\}$ and for all $e \in E$, $f_e(x) \neq 1$. This is a trick to merge two kinds of judgments, one is to check whether an encountered vertex has been investigated before and the other is to decide whether its value should be overridden or not. In general, the element 1 is artificially introduced and thus this assumption can be satisfied without losing generality.

**Output** Let $V_1$ be a subset of $V$ that are reachable from $s$ and $V_2$ be a subset that are not reachable, then for $v \in V_1$, $x_v$ satisfies the equations (1) and for $v \in V_2$, $x_v = 1$.

**Internal data structure** a set $S$ to retain vertices

**Procedure**

```
For each $v \in V$, $x_v \leftarrow 1$
$x_s \leftarrow b_s$
$S \leftarrow \{s\}$
While $S \neq \emptyset$ do
    Take an arbitrary vertex $v$ out of $S$.
    For each $e \in$ out($v$) do
        If $\neg x_{t(e)} \leq f_e(x_v)$ then
            $x_{t(e)} \leftarrow x_{t(e)} \wedge f_e(x_v)$
            Add $t(e)$ to $S$.
```

To describe the above algorithm in CafeOBJ, we need two recursive functions, "update" and "iterate", corresponding to the two **for** loops that appear in the informal algorithm description. Also, we have to define data structures that maintain the set of vertices $S$ and the table relating each vertex $v$ with its value $x_v$. Writing the algorithm in CafeOBJ based on these policies is almost straightforward.

### 4.2. Algorithm in CafeOBJ

Based on the policies stated above, the iterative algorithm can be described in CafeOBJ as follows.

Firstly, the data structure relating a vertex to its value is defined.

```
module! PAIR[X1 :: TRIV, X2 :: TRIV] {
  [ Pair ]
  signature {
    op (_,_) : Elt.X1 Elt.X2 -> Pair
  }
}
module! XV[G :: GRAPHEX, L :: LATTICE-T]
{ protecting(PAIR(X1 <= view to
                G { sort Elt -> Vertex },
                  X2 <= view to
                L { sort Elt -> Elt })
             * { sort Pair -> Xv })
}
module! XVFUNC[G :: GRAPHEX,
               L :: LATTICE-T] {
  protecting (LIST[X <= view to
              XV[G,L] {sort Elt -> Xv}]
              * { sort List -> XVfunc })
  signature {
    op _<_> : XVfunc Vertex -> Elt
  }
  axioms {
    var xf : XVfunc
    vars u, v : Vertex
    var e : Elt
    ceq (( u , e ) xf) < v >
         = e if u == v .
    ceq (( u , e ) xf) < v >
         = xf < v > if u =/= v .
  }
}
```

The global state of the algorithm is maintained by a record type data consisting of the set of vertices $S$, represented by a vertex list `vlist` and the association list of vertex-value pairs `xf`.

```
module* GFPSTATE[G :: GRAPHEX,
                 L :: LATTICE-T] {
  protecting (XVFUNC[G,L])
  signature {
    record State {
     vlist : Vlist
     xf    : XVfunc
    }
  }
}
```

Now, the algorithm can be described using the above defined data structures and two recursive functions `update` and `iterate`.

```
module! GFPALG[G :: GRAPHEX,
               L :: LATTICE-T] {
  protecting (FIXEDPOINT[G,L])
  protecting (GFPSTATE[G,L])
  signature {
    op initxv : -> XVfunc
    op solve : -> XVfunc
    op iterate : State -> XVfunc
    op update : Vertex XVfunc Elist Vlist
                -> State
  }
  axioms {
    var v : Vertex
    var vl : Vlist
    var e : Edge
    var el : Elist
    var z : XVfunc
    eq solve = iterate(State{
                    vlist = (source nil),
                    xv = initxv }) .
    eq iterate(State{ vlist = (v vl),
                      xv = z })
        = iterate(State{
           vlist = (vl
                    vlist(update(v, z,
                       outedge(v), nil))),
           xv = xv(update(v, z,
                    outedge(v), nil)) }) .
    eq iterate(State{ vlist = nil,
                      xv = z} ) = z .
    ceq update(v, z, (e el), vl)
         = update(v,
                  ( tail(e) ,
                   (z < tail(e) >) ^
                   (fe(e) < (z < v >) >)),
                  e, (tail(e) vl))
        if (z < tail(e) >) <=
                    (fe(e) < (z < v >) >)
           =/= true .
    ceq update(v, z, (e el), vl)
         = update(v, z, e, vl)
        if (z < tail(e) >) <=
                    (fe(e) < (z < v >) >)
           == true .
    eq update(v, z, nil, vl)
        = State{ vlist = vl, xv = z } .
  }
}
```

## 5. Discussions

Compared to the description we made in HOL [13], the description in CafeOBJ is more human-friendly, i.e. easy to read and comprehend and also crisp. However, the objective of the use of HOL was to give a mechanical proof of the algorithm, whereas the use of CafeOBJ this time is to write a precise problem specification and solution algorithm and also to show the way of customizing the general description to fit into specific problem domains. In principle, we can execute proof in CafeOBJ just as we did in HOL but the tool support for mechanical proof in CafeOBJ at the moment is not so strong as that in HOL. We did execute a kind of program simulation by using the reduction mechanism to check the execution of the algorithm but that does not completely correspond to the verification in HOL that showed the algorithm satifies the specification.

There have been some works of using CafeOBJ for writing real-world object-oriented software specifications [6, 12]. Compared to the ODP trader application reported in the paper [6], our approach covers wider range of applications but neglects implementation details. We believe attemps of using formal specification languages for writing real-world applications like these should be continuted from various aspects.

The computation model contained in the algorithm stated above is different from the one interpreting the equations in FIXEDPOINT by the term rewriting scheme. In the term rewriting scheme, computation proceeds by replacing a subterm of a given expression that matches to a left hand side of some equation by its right hand side, until a canonical term is obtained. In our algorithm, all the left hand sides of the equations are variables and values (or canonical terms) are substituted to the variables of the right hand sides of the equations to produce new values for the corresponding left hand side variables. This process is continued until all the values for the variables converge.

There is another class of algorithms, viz. elimination type [9]. Elimination algorithms proceed from left to right just like term rewriting, eliminating exactly one variable by each substitution. Eventually one variable remains and a value can be assigned to this variable. Then, the computation proceeds "backward" and a value is assigned to variables one by one.

It might be interesting to further analyze this relation between term rewriting computation vs. iterative algorithms and elimination algorithms.

## References

[1] CoFI Language Design Task Group. Casl – the cofi algebraic specification language, October 1998.

[2] R. Diaconescu and K. Futatsugi. *CafeOBJ Report — The Langauage, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998.

[3] E. A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B*, chapter 16. Elsevier Science publishers B.V., 1990.

[4] A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.

[5] B. Meyer. The next software breakthrough. *IEEE Computer*, pages 113–114, July 1997.

[6] S. Nakajima. Using algebraic specification techniques in development of object-oriented frameworks. In *FM'99 – Formal Methods*, volume 1709 of *LNCS*, pages 1664–1683. Springer, September 1999.

[7] M. Phillips. Cics/esa 3.1 experiences. In J. E. Nicholls, editor, *Z User Workshop: Proceedings of the Fourth Annual Z User Meeting, Oxford 1989*, pages 179–185. Springer-Verlag, 1990.

[8] T. Reps. Program analysis via graph reachability. In *Proc. 1997 Int. Logic Prog. Symp.*, October 1997.

[9] T. Tamai. The structure of elimination-type algorithms for a class of fixed-point problems on graphs. *Electronics and Communications in Japan— Part III: Fundamental Electronic Science*, 77(2):1–13, 1994.

[10] T. Tamai. A class of fixed-point problems on graphs and iterative solution algorithms. In A. Pnueli and H. Lin, editors, *Logic and Software Engineering*, pages 102–121. World Scientific, 1996.

[11] J. M. Wing. A specifier's introduction to formal methods. *IEEE Computer*, pages 8–24, September 1990.

[12] M. Wirsing and A. Knapp. A formal approach to object-oriented software engineering. In *Proc. 1st Workshop on Rewriting Logic and its Applications*, 1996.

[13] M. Yamamoto, K. Takahashi, M. Hagiya, S. Nishizaki, and T. Tamai. Formalization of graph search algorithms and its applications. In *Theorem Proving in Higher Order Logics*, volume 1479 of *LNCS*, pages 479–496. Springer, 1998.