An Exercise in the Formal Derivation of Parallel Programs: Maximum Flows in Graphs

EDGAR KNAPP
The University of Texas at Austin

We apply a new method for the development of parallel programs to the problem of finding maximum flows in graphs. The method facilitates concurrent program design by separating the concerns of correctness from those of hardware and implementation. It uses predicate transformer semantics to define a set of basic operators for the specification and verification of programs. From an initial specification program development proceeds by a series of refinement steps, each of which constitutes a strengthening of the specification of the previous refinement. The method is completely formal in the sense that all reasoning steps are performed within predicate calculus. A program is viewed as a mathematical object enjoying certain properties, rather than in terms of its possible executions. We demonstrate the usefulness of the approach by deriving an efficient algorithm for the Maximum Flow Problem in a top-down manner.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages, methodologies*; D.2.2 [**Software Engineering**]: Tools and Techniques—*structured programming, top-down programming*; D.2.4 [**Software Engineering**]: Program Verification—*correctness proofs*; D.3.3 [**Programming Languages**]: Language Constructs—*concurrent programming structures*; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*alternation and nondeterminism, parallelism*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, invariants, logics of programs, specification techniques*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*graph algorithms, network problems*

General Terms: Algorithms, Design, Languages, Theory, Verification

Additional Key Words and Phrases: Derivation of algorithms, distributed algorithms, maximum flow, program development, stepwise refinement, UNITY

INTRODUCTION

We use a novel method for the development of parallel programs to derive an efficient algorithm for the Maximum Flow Problem. The formalism we use is named UNITY, and was developed by Chandy and Misra [1]. It is new in the

This work was supported in part by the Office of Naval Research under contract ONR N0014-86-K-0763. Author's current address: 3601-C North Hills Drive, Austin, TX 78731-3043.

sense that it abandons the notion of a process as a fundamental concept of parallel program design and that it facilitates program derivation by rigorously separating the concerns of program correctness from those of hardware and implementation (e.g., architectures, synchronization primitives, etc.). The method is also completely formal in the sense that operational reasoning is eliminated entirely; all inferences are done within predicate calculus. Furthermore, a program is viewed as a mathematical object enjoying certain properties (e.g., invariants, stability), and not in terms of its possible executions.

Program development in UNITY proceeds by stepwise refinement. While most methodologies of this type have programs as the object of refinement, our approach consists of the refinement of specifications.

UNITY lends itself naturally to both procedural types of refinement (e.g., the realization of a particular solution strategy into an algorithm) and to data refinement (e.g., replacing a global data structure by a distributed one). In this paper our focus is on the former; an example of the latter can be found in [13].

The goal of program development, the program proper, is only obtained in the last step of this development process. This is typically done when the refined specification is restrictive enough to suggest a translation into UNITY code.

As a consequence of this approach, proving the correctness of a refinement becomes an exercise in logic, and is therefore simpler than proving two programs equivalent, or that each in a series of refined programs meets a particular specification.

We use the Maximum Flow Problem as an example of the usefulness of the UNITY approach. The algorithm we are deriving is originally due to Goldberg and Tarjan [5, 6]. Starting from the problem specification, we derive a series of top-down refinements, proving at each stage that the correctness of our solution is preserved. Our refinement steps are motivated mostly by heuristics about the syntactic shape of our formulas, that is, we strive to eliminate formulas that are not easily manipulatable symbolically. For example, one of our refinement steps consists of replacing reasoning about paths by reasoning about a function. In addition, our development clearly shows which parts of the problem can be solved routinely and which require more insight. The algorithm we will thus derive is a generic version of—according to Goldberg and Tarjan—the most efficient algorithms known today for both sequential and parallel architectures.

Our work differs from most others in the area of parallel program verification (cf., [9–12]) in several important respects. First of all, unlike temporal logics and I/O automata, the UNITY formalism is independent from an operational model of program execution, even though operational notions such as execution sequences or histories may serve as the intuition behind UNITY logic. As a consequence, fairness is not an issue: it is built into the **ensures** operator, the basic notion of progress in UNITY. Second, our emphasis is not on a posteriori verification of programs, that is, demonstrating that a given program meets its specification, but on constructively deriving a program from its specification by a succession of refinement steps. In this sense UNITY is very much in the spirit of "verifying a program before it is written," a paradigm advocated for sequential programming most notably by Dijkstra [2] and Gries [7].

Our paper is organized as follows: Section 1 contains an introduction to our notation. The first part of this section presents some general notational

conventions; the second contains a brief discussion of the proof format we will be using, while the third part presents that subset of UNITY that is needed for our refinements. This presentation of UNITY is necessarily incomplete, and the reader is referred to [1] for an exhaustive treatment. The last part of this section formalizes our notion of refinement of specifications.

Section 2 defines the Maximum Flow Problem formally. The specification we obtain is the starting point for a series of stepwise refinements. These refinements are presented in Section 3; they lead to a set of about ten properties, from which, in Section 4, the program text is derived quite straightforwardly. A discussion of the advantages and disadvantages of this approach concludes our paper.

1. NOTATION

1.0 Notational Conventions

We use the following notational conventions: quantified expressions are written in the format

$$\langle \oplus x : r.x : t.x \rangle,$$

where $\oplus$ is any associative and symmetric operator. Furthermore, x is called the *dummy*, $r.x$ is called the *range*, and $t.x$ the *term* of the quantification. In case $\oplus$ has an identity element e, $\langle \oplus x : false : t.x \rangle$ is well defined and equal to e. If in a quantified formula the range is omitted, quantification is over all elements of the domain of the dummy.[1]

For example, $\langle \forall i : 0 \leq i \wedge i < n : A.i > 0 \rangle$ holds if and only if all elements of A in the range between 0 and $n - 1$ are positive. Likewise $\langle +i :: A.i \rangle$ denotes the sum of all elements of A; in the latter example, the range of i is understood from the given context.

The operators we use are summarized below, ordered by binding powers, with the top row having the weakest and the bottom row having the strongest binding power. Precise definitions of all new operators are given later in this section and in Section 1.2.

$$\equiv$$
$$\Leftarrow, \Rightarrow,$$
$$\textbf{unless, stable, invariant, ensures}, \mapsto,$$
$$\wedge, \vee,$$
$$\neg,$$
$$=, \neq, <, \geq, >, \leq, \succ, \in,$$
$$\min, +, -, \text{path}$$

The binary infix operator min is defined as:

$$a = b \min c \equiv (a = b \vee a = c) \wedge a \leq b \wedge a \leq c.$$

It is symmetric, idempotent, and associative.

In a graph G, we define a predicate "path" on vertex pairs as:

$$x \text{ path } y \equiv \text{there is a path from } x \text{ to } y \text{ in } G.$$

[1] When the term of the quantification is a predicate, we will write $\forall$ instead of $\wedge$ and $\exists$ instead of $\vee$.

We use the fact that the relation "path" is reflexive and transitive.

The value of the expression $\langle t \textbf{ if } b \rangle$, where t is an integer expression, is defined to be t in case b holds, and 0 otherwise.

1.1 Proof Format

Most of our proofs are purely calculational in the sense that they consist of a number of syntactic transformations instead of semantic reasoning steps. We use a proof format that was first proposed by Feijen, Dijkstra, and others, and that greatly facilitates this kind of proof.

For example, a proof that $A \equiv D$ may be rendered in our format as

$$
\begin{array}{ll}
 & A \\
= & \{\textit{hint why } A \equiv B\} \\
 & B \\
= & \{\textit{hint why } B \equiv C\} \\
 & C \\
= & \{\textit{hint why } C \equiv D\} \\
 & D
\end{array}
$$

We also allow other transitive operators in the leftmost column. Among these are the more traditional **implies** ($\Rightarrow$), but also, for reasons of symmetry, **follows-from** ($\Leftarrow$). It turns out that the latter is more than a mere convenience: proofs that strike the reader as requiring considerable clairvoyance when presented in one direction, when written the other way round, have the pleasant property that each manipulation is strongly suggested by what was done previously.

We leave this discussion at these very general remarks. For a more thorough treatment of this subject, the reader is referred to [3].

1.2 An Introduction to UNITY

Parallel program design in UNITY is the subject of a recent book [1]. We describe that subset of UNITY that is relevant to our discussion; footnotes identify the differences to full UNITY, as described in [1].

Programs in UNITY

A UNITY program consists of four parts:

(0) the **declare**-section, or **declare** for short, contains a series of Pascal-style declarations;

(1) the **initially**-section, or **initially** for short, is a set of equations defining the initial values for some or all the program variables;

(2) the **always**-section, or **always** for short, contains a set of defining equations for a subset of program variables; there are certain restrictions on the form these equations may assume;

(3) the **assign**-section, or **assign** for short, consists of a finite nonempty set of multiple assignment statements.

The purpose of the first two sections should be quite obvious. Each equation in **always** defines the value of some variable. The restrictions on the equations serve to ensure that circular and conflicting definitions are excluded. These

restrictions can be formalized quite straightforwardly. The interested reader is referred to [1]. One property of the **always**-section worth noting is that each equation in it is a program invariant.

Let x denote a list of program variables, and $f.x$ a list of expressions, possibly depending on x and matching the variables in x in number and type. The **assign**-section contains statements of the form

$$x := f.x \textbf{ if } B$$

where $x := f.x$ is a multiple assignment statement and B is a Boolean expression.[2] With **wp** standing for Dijkstra's weakest precondition, we define the predicate transformer semantics of such a statement as follows:

$$\textbf{wp}(\text{“}x := f.x \textbf{ if } B\text{”}, Q) \equiv (B \Rightarrow \textbf{wp}(\text{“x := } f.x\text{”}, Q)) \wedge (\neg B \Rightarrow Q). \quad \text{(D0)}$$

The semantics of the multiple assignment $x := f.x$ are

$$\textbf{wp}(\text{“}x := f.x\text{”}, Q) \equiv Q[x := f.x] \quad \text{(D1)}$$

where $Q[x := f.x]$ stands for Q with all occurrences of variables in x simultaneously replaced by the matching expressions in $f.x$.

As an example consider the statement

$$x, y := y, x \textbf{ if } x < y.$$

To derive the weakest precondition such that this statement establishes the predicate $x > y$ we calculate

$$\begin{aligned}
&\textbf{wp}(\text{“}x, y := y, x \textbf{ if } x < y\text{”}, x > y) \\
&= \{\textit{definition } \textbf{wp}\} \\
&\quad (x < y \Rightarrow \textbf{wp}(\text{“}x, y := y, x\text{”}, x > y)) \wedge (x \geq y \Rightarrow x > y) \\
&= \{\textit{definition } \textbf{wp}, \textit{simplify second conjunct}\} \\
&\quad (x < y \Rightarrow y > x) \wedge x \neq y \\
&= \{\textit{first conjunct simplifies to true}\} \\
&\quad x \neq y
\end{aligned}$$

Equations in **initially** and **always** and statements in **assign** are separated by the operator □. This operator is symmetric, associative, and idempotent. When used to separate equations, its semantics are those of logical conjunction ($\wedge$). In the **assign**-section it serves as a statement separator, and its identity element is the empty assignment, commonly denoted by **skip**.

From an operational point of view, we can think of a UNITY program as being executed by repeating the following, *ad infinitum*: select any statement from **assign** and execute it. The only requirement we impose on the selection process is that no statement in the set be ignored forever. A variable defined in **always** can be thought of as being evaluated when it is accessed by the program.

As a consequence, UNITY programs do not terminate in the conventional sense. We do however stipulate that each individual UNITY statement terminate. Formally, this is expressed by

$$\langle \forall S :: \textbf{wp}(S, \textit{true}) \equiv \textit{true} \rangle. \quad \text{(D2)}$$

[2] Full UNITY has a slightly more general form of multiple assignment.

The Specification Language

Problem specifications in UNITY are written using three basic binary operators: **unless**, **ensures**, and $\mapsto$ ("leads to"). Using the predicate transformer semantics of the individual statements in **assign**, we define

$$P \textbf{ unless } Q \equiv \langle \forall S : S \in \textbf{assign} : P \wedge \neg Q \Rightarrow \textbf{wp}(S, P \vee Q) \rangle. \qquad \text{(D3)}$$

Intuitively, P **unless** Q means that whenever predicate P holds for a program state, Q holds in this state, or P continues to hold at least until Q holds. Note that this allows for the case that Q never holds; in such a case P continues to hold forever.

The following property of **unless** follows directly from its definition:

$$(P \Rightarrow Q) \Rightarrow P \textbf{ unless } Q.$$

As an example of an **unless** property, consider the specification that "a hungry philosopher remains hungry until he eats (if ever)." Formally,

$$\langle \forall phil :: phil.hungry \textbf{ unless } phil.eats \rangle.$$

There is a special case of **unless** that deserves special attention, because it occurs quite frequently. Consider the property P **unless** *false*. Going back to the definition of **unless**, we find that this specifies that each statement preserves P. We call such a predicate a **stable** property, and define

$$\textbf{stable } P \equiv P \textbf{ unless } false. \qquad \text{(D4)}$$

An example of a stable property is the fact that *sent.ch*, the number of messages sent along channel *ch*, does not decrease. We write this as

$$\langle \forall k :: \textbf{stable } sent.ch \geq k \rangle.$$

Another concept that turns out to be of great importance is that of an **invariant**. Its definition is straightforward:

$$\textbf{invariant } P \equiv (\textbf{initially} \Rightarrow P) \wedge \textbf{stable } P. \qquad \text{(D5)}$$

An example of an invariant property is the requirement that neighboring philosophers do not eat simultaneously, i.e.,

$$\textbf{invariant } \langle \forall phil0, phil1 :: \neg(phil0.eats \wedge phil1.eats \wedge phil0 \text{ neighbor } phil1) \rangle.$$

The reader may have noticed that so far the properties one can specify are limited to *safety* properties, i.e., properties that disallow certain transitions between program states. When specifying concurrent programs, however, we are often interested in stating that a certain predicate holds at some point in the future. In UNITY, this requirement is expressed using $\mapsto$; informally, $P \mapsto Q$ states that if P holds in some state of the program, then eventually a state is reached in which Q holds.

As an example, consider the requirement that a hungry philosopher eat eventually. We write this as

$$\langle \forall phil :: phil.hungry \mapsto phil.eats \rangle.$$

Next we want to investigate this new operator more closely. What properties should $\mapsto$ have? Clearly, we want it to be transitive, i.e.,

$$(P \mapsto Q) \wedge (Q \mapsto R) \Rightarrow P \mapsto R. \tag{D6}$$

Furthermore, we have to supply a method to prove a $\mapsto$ property from the text of a program. In order to do this, we define an operator **ensures**, using predicate transformer semantics in the following way:

$$\begin{aligned} &P \textbf{ ensures } Q \\ &\quad \equiv (P \textbf{ unless } Q) \wedge \langle \exists S : S \in \textbf{assign} : P \wedge \neg Q \Rightarrow \textbf{wp}(S, Q) \rangle. \end{aligned} \tag{D7}$$

We refer to the two conjuncts in the above definition as the **unless** part and the existential part, respectively. D7 formalizes the idea that, since no statement in **assign** is ignored forever, some S that establishes Q will be executed eventually in a state satisfying P. Note the following property of **ensures**:

$$(P \Rightarrow Q) \Rightarrow P \textbf{ ensures } Q.$$

Using **ensures**, we can now continue with the definition of $\mapsto$:

$$P \textbf{ ensures } Q \Rightarrow P \mapsto Q. \tag{D8}$$

This states that every **ensures** property is a $\mapsto$ property. We, therefore, immediately observe that

$$(P \Rightarrow Q) \Rightarrow P \mapsto Q.$$

In particular, we get, using the transitivity of $\mapsto$:

LEMMA 0 (Strengthening lhs, weakening rhs of $\mapsto$).

$$\begin{aligned} &(P \Rightarrow Q) \wedge (Q \mapsto R) \Rightarrow P \mapsto R \\ &(P \mapsto Q) \wedge (Q \Rightarrow R) \Rightarrow P \mapsto R \end{aligned}$$

To conclude our definition of $\mapsto$, we given the following induction principle[3] for $\mapsto$: let M be a function from the program state to a well-founded set with partial order $\succ$,

$$\langle \forall k :: P \wedge M = k \mapsto M \succ k \rangle \Rightarrow \mathit{true} \mapsto \neg P. \tag{D9}$$

Theorems About UNITY Operators

Using the operators we just defined, we can now prove a number of useful results. They will come in handy when we refine specifications and reason about the correctness of parallel programs. The theorems are given here without proof. For detailed proofs, the interested reader is referred to [1].

THEOREM 0 (Finite Disjunction). *For any finite index set I:*

$$\langle \forall i : i \in I : P.i \mapsto Q.i \rangle \Rightarrow \langle \exists i : i \in I : P.i \rangle \mapsto \langle \exists i : i \in I : Q.i \rangle.$$

THEOREM 1 (PSP).

$$(X \mapsto Y) \wedge (W \textbf{ unless } Z) \Rightarrow X \wedge W \mapsto (Y \wedge W) \vee Z.$$

[3] Reference [1] contains a more general form of induction principle.

The acronym PSP stands for progress, safety, progress, since from a progress and a safety property we derive another progress property.

1.3 Refinement of Specifications

We now formalize our notion of refinement. Let P, Q be specifications (e.g., sets of UNITY properties). We say that P is a refinement of Q if and only if $P \Rightarrow Q$.

Each of the refinement steps of Section 3 will be of this form: a set of UNITY properties is replaced by another that is shown to be logically stronger.

2. A SPECIFICATION OF THE MAXIMUM FLOW PROBLEM

We consider a set V of n vertices, $n \geq 2$, which includes two distinguished vertices s and t, called *source* and *sink*, respectively. All other vertices are *internal* vertices, and their entirety is denoted by V^-. Associated with each ordered pair $(v, w) \in V \times V$ is a finite real-valued *capacity* $c(v, w) \geq 0$.

The problem is to compute a real-valued function f on vertex pairs that satisfies the constraints given below. For each vertex pair (v, w), $f(v, w)$ is called the *direct flow* from v to w.

The first constraint we impose is that the direct flow from v to w be limited by the capacity $c(v, w)$, for all $v, w \in V$. Formally,

$$\langle \forall v, w :: f(v, w) \leq c(v, w) \rangle. \tag{S0}$$

To simplify the formal treatment of the problem somewhat, we stipulate that direct flows between the same vertex pair, but in opposite directions, be equal in magnitude but opposite in sign, i.e.,

$$\langle \forall v, w :: f(v, w) = -f(w, v) \rangle. \tag{S1}$$

Note that S1 implies $\langle \forall v :: f(v, v) = 0 \rangle$. For a vertex v, we define the *net flow* into v to be $e.v = \langle +x :: f(x, v) \rangle$. We require that this net flow vanish for all internal vertices. This gives us Kirchhoff's Law:

$$\langle \forall v : v \in V^- : e.v = 0 \rangle. \tag{S2}$$

A *flow* is a function f satisfying the *flow conditions* S0, S1, and S2. The *value* of a flow is defined to be $e.t$, the net flow into the sink. A *maximum flow* is a flow with the maximum value, i.e., one for which

$$e.t \text{ is maximum.} \tag{S3}$$

A maximum flow can be shown to exist in all cases (cf., [0]). We say that our solution is correct if our program establishes $true \mapsto$ S0 $\wedge$ S1 $\wedge$ S2 $\wedge$ S3.[4]

3. STEPWISE REFINEMENT OF THE SPECIFICATION

3.0 A First Solution

After stating the problem formally, our first design task is to decide on a general solution strategy. Our design decision is to require that S0 and S1 be invariants of our program, giving us properties P0 and P1 below. S2 seems to be far too

[4] The reader may ask why we do not require *stable* S0 $\wedge$ S1 $\wedge$ S2 $\wedge$ S3. This was omitted for simplicity. However, the final program indeed satisfies this additional requirement.

restrictive to serve as an invariant. Therefore, we only require that S2 be established eventually (giving us P2), and that upon its establishment S3 hold (giving us P3).

We summarize the properties of our program below and note that the correctness of the solution is immediate.

$$\textbf{invariant } \langle \forall v, w :: f(v, w) \leq c(v, w) \rangle. \tag{P0}$$
$$\textbf{invariant } \langle \forall v, w :: f(v, w) = -f(w, v) \rangle. \tag{P1}$$
$$\textit{true} \quad \mapsto \langle \forall v : v \in V^- : e.v = 0 \rangle. \tag{P2}$$
$$\textbf{invariant } \langle \forall v : v \in V^- : e.v = 0 \rangle \Rightarrow e.t \text{ is maximum}. \tag{P3}$$

3.1 Refinement 0: Preflow

Our first refinement is motivated by the notion of preflow introduced in [8]. A *preflow* is a function f satisfying S0, S1, and the following modified form of S2:

$$\langle \forall v : v \neq s : e.v \geq 0 \rangle. \tag{S4}$$

Whereas S2 was considered too strong a condition to be eligible as an invariant, our design decision is to require the invariance of S4. So we strengthen our specification by adding

$$\textbf{invariant } \langle \forall v : v \neq s : e.v \geq 0 \rangle. \tag{P4}$$

Since this addition constitutes a strengthening of our specification, the correctness of our solution is preserved trivially. Henceforth, f in general refers to a preflow.

3.2 Refinement 1: Residual Capacity

Looking at P3 we realize that the rhs of the implication is not quite in a form amenable to formal manipulation. Our next step is therefore to replace P3 by a condition that can be manipulated more easily. To this end, we define the *residual capacity* $r(v, w)$ of a vertex pair (v, w) to stand for the "unused" portion of the capacity $c(v, w)$, formally $r(v, w) = c(v, w) - f(v, w)$.

Now consider the *residual graph* R with vertex set V and the set of directed edges consisting of all vertex pairs (v, w) such that $r(v, w) > 0$. Note that R depends on f. Paths in R and flows are related by the following classical result [4]:

THEOREM 2. *For a flow f: ($e.t$ is maximum $\equiv \neg(s$ path $t)$).*

Remark. We are considering only paths in the residual graph R. Therefore, whenever we refer to paths, we mean paths in R.

We now replace P3 by

$$\textbf{invariant } \neg(s \text{ path } t). \tag{P5}$$

Observing that

$$\begin{array}{l} \langle \forall v : v \in V^- : e.v = 0 \rangle \wedge \neg(s \text{ path } t) \\ \Rightarrow \{\textit{by Theorem 2, since } f \textit{ is a flow}\} \\ \quad e.t \text{ is maximum} \end{array}$$

establishes that this replacement preserves the correctness of our solution, and concludes this refinement step.

3.3 Refinement 2: Altitudes

Our specification at this point consists of properties P0, P1, P2, P4, and P5. Of all these, the last one is the most difficult to manipulate formally, since it involves paths. Let us therefore concentrate on how to refine P5. To this end, imagine each node being endowed with an additional attribute, a natural number, which we call the altitude of the node. Net flow is transferred only along downward edges in R, i.e., from nodes with higher altitudes to nodes of lower altitude. We arbitrarily choose the altitude of the sink equal to zero. The altitudes of internal vertices may change. For the time being, we leave open the choice for the altitude of the source. Formally, with $alt.x$ denoting the altitude of node x, we propose:

$$\langle \forall v :: alt.v \geq 0 \rangle \wedge alt.t = 0. \tag{S5}$$

Our goal now is to replace reasoning about paths (as in P5) by reasoning about altitudes. To this end, we require for an edge (v, w) in the residual graph v's altitude not to exceed w's altitude by more than one, formally:

$$\langle \forall v, w : r(v, w) > 0 : alt.v \leq alt.w + 1 \rangle. \tag{S6}$$

To see how S6 can be used to maintain P5, consider a simple path $p = v_0 v_1 \cdots v_l$ in R. Note that $l < n$, since there are at most n distinct vertices in p. Now we observe that for any such p:

$$\begin{array}{ll} & p \text{ is a path in } R \\ \Rightarrow & \{\textit{definition of } R\} \\ & \langle \forall i : 0 \leq i < l : r(v_i, v_{i+1}) > 0 \rangle \\ \Rightarrow & \{S6\} \\ & \langle \forall i : 0 \leq i < l : alt.v_i \leq alt.v_{i+1} + 1 \rangle \\ \Rightarrow & \{\textit{arithmetic}\} \\ & alt.v_0 \leq alt.v_l + l \\ \Rightarrow & \{\textit{since } l < n\} \\ & alt.v_0 < alt.v_l + n \end{array}$$

Since there is a path between a pair of vertices in a graph if and only if there is a simple path between them, we have proved the following:

LEMMA 1. $\langle \forall v, w :: v \textit{ path } w \Rightarrow alt.v < alt.w + n \rangle$.

From this we conclude that

$$\begin{array}{ll} & \neg(s \text{ path } t) \\ \Leftarrow & \{\textit{instantiate the contrapositive of Lemma 1 with } v, w := s, t\} \\ & alt.s \geq alt.t + n \\ = & \{\textit{since } alt.t = 0\} \\ & alt.s \geq n \end{array}$$

This means that in order to guarantee P5, all we have to do is to place the source high enough! That is, $alt.s \geq n$ will do. Summing up, we replace P5 by

$$\textbf{invariant } \langle \forall v :: alt.v \geq 0 \rangle \wedge alt.t = 0 \wedge alt.s \geq n. \tag{P6}$$

$$\textbf{invariant } \langle \forall v, w : r(v, w) > 0 : alt.v \leq alt.w + 1 \rangle. \tag{P7}$$

3.4 Refinement 3: Progress

After three refinement steps, our specification now consists of the properties P0, P1, P2, P4, P6, and P7. Next we want to investigate what properties the altitude function should have in order to guarantee progress. So our goal in this refinement is to replace P2 by a series of properties.

What constitutes progress for our program? Observe that eventually all net flows of internal vertices vanish (by P2). So we have to aim at reducing net flows of internal vertices. We call an internal vertex v with a positive net flow a *surplus* vertex, and define

$$sur.v \equiv v \in V^- \wedge e.v > 0.$$

Recall that we imposed the requirement that net flow be transferred only along downward edges in R. We call a vertex without any outgoing downward edges a *disabled* vertex, i.e.,

$$dis.v \equiv \langle \forall w : r(v, w) > 0 : alt.v \leq alt.w \rangle.$$

We call a vertex *enabled* if it is not disabled. Formally, we derive:

$$\begin{aligned} & \neg dis.v \\ = & \{definition\ dis.v,\ de\ Morgan\} \\ & \langle \exists w : r(v, w) > 0 : alt.v > alt.w \rangle \\ = & \{P7\ and\ arithmetic\} \\ & \langle \exists w : r(v, w) > 0 : alt.v = alt.w + 1 \rangle \end{aligned}$$

Progress can be made in two ways: either (0), by an increase in altitudes, or (1), by a change in net flows.

Ad (0): We require that if there is a disabled surplus vertex, then some altitude with increase eventually. If we also stipulate that altitudes do not decrease, an appropriate measure of progress made by raising altitudes is any expression that is increasing in each altitude. The simplest such expression we can think of is $a = \langle +v :: alt.v \rangle$. So we obtain the requirements:

$$a = k \textbf{ unless } a > k. \tag{P8}$$

$$\langle \exists v : sur.v : dis.v \rangle \wedge a = k \ \mapsto \ a > k. \tag{P9}$$

Ad (1): Now consider the case in which there exists an enabled surplus vertex. Let b be an expression that characterizes progress made by net flow transfer. Since there does not yet seem to be an obvious choice for b, we postpone the construction of b. So we require that our program meet

$$\langle \exists v : sur.v : \neg dis.v \rangle \wedge b = l \ \mapsto \ b > l. \tag{P10}$$

Proof of Correctness of this Refinement

Let us now see how the three properties we proposed above (viz., P8, P9, and P10) allow us to replace P2. Towards this end, we first combine P8 and P10 using the PSP-Theorem, abbreviating $ENA \equiv \langle \exists v : sur.v : \neg dis.v \rangle$ and yielding,

$$ENA \wedge a = k \wedge b = l \mapsto (a = k \wedge b > l) \vee a > k.$$

We combine this with P9, using the Finite Disjunction Theorem (abbreviating in P9 $DIS \equiv \langle \exists v : sur.v : dis.v \rangle$):

$$
\begin{array}{ll}
 & (ENA \wedge a = k \wedge b = l) \vee (DIS \wedge a = k) \mapsto (a = k \wedge b > l) \vee a > k \vee a > k \\
\Rightarrow & \{\textit{strengthen lhs of} \mapsto, \textit{simplify rhs}\} \\
 & (ENA \wedge a = k \wedge b = l) \vee (DIS \wedge a = k \wedge b = l) \mapsto (a = k \wedge b > l) \vee a > k \\
= & \{\textit{predicate calculus}\} \\
 & (ENA \vee DIS) \wedge a = k \wedge b = l \mapsto (a = k \wedge b > l) \vee a > k
\end{array}
$$

At this point, the introduction of a metric $M = (a, b)$ with lexicographic order $\succ$ suggests itself. Observe that in order for $(M, \succ)$ to form a well-founded set, we have to require that a and b be bounded from above.

Before we continue with the derivation, we first simplify $ENA \vee DIS$:

$$
\begin{array}{ll}
 & ENA \vee DIS \\
= & \{\textit{definitions } ENA \textit{ and } DIS\} \\
 & \langle \exists v : sur.v : \neg dis.v \rangle \vee \langle \exists v : sur.v : dis.v \rangle \\
= & \{\textit{combining terms}\} \\
 & \langle \exists v : sur.v : true \rangle \\
= & \{\textit{definition of } sur.v \textit{ and trading}\} \\
 & \langle \exists v : v \in V^- : e.v > 0 \rangle
\end{array}
$$

Now we resume our former calculation, using $M = (k, l) \equiv a = k \wedge b = l$ and $M \succ (k, l) \equiv (a = k \wedge b > l) \vee a > k$:

$$
\begin{array}{ll}
 & \langle \exists v : v \in V^- : e.v > 0 \rangle \wedge M = (k, l) \mapsto M \succ (k, l) \\
\Rightarrow & \{D9, \textit{induction on} \mapsto, \textit{provided } a \textit{ and } b \textit{ are bounded from above}\} \\
 & true \mapsto \neg \langle \exists v : v \in V^- : e.v > 0 \rangle \\
= & \{\textit{de Morgan}\} \\
 & true \mapsto \langle \forall v : v \in V^- : e.v \leq 0 \rangle \\
= & \{P4\} \\
 & true \mapsto \langle \forall v : v \in V^- : e.v = 0 \rangle
\end{array}
$$

This establishes that P2 can be replaced by P8, P9, and P10, provided a and b are bounded from above. So we are left with investigating an appropriate choice for b and proving bounds on a and b. Therefore, let us now look into some consequences of our refinement so far.

3.5 Summary and Consequences of Our Design Decisions

In the course of our refinements, the following properties have emerged: P0, P1, P4, P6, P7, P8, P9, and P10. We now prove the boundedness of a. Since by P6, a is bounded from below by 0, we are left with establishing an upper bound for altitudes.

Let v be a fixed surplus vertex. Define the predicate H on vertices as follows:

$$H.x \equiv v \text{ path } x.$$

Observe that by the reflexivity of path, $H.v$ holds. Furthermore, note that by the transitivity of path, we have

$$\langle \forall x, y : H.x \wedge \neg H.y : \neg x \text{ path } y \rangle.$$

Our first goal is to establish $H.s$. Towards this end, we observe for any x and y such that $H.x$ and $\neg H.y$:

$$
\begin{array}{ll}
 & 0 \\
\geq & \{\textit{by above observation, } \neg x \textit{ path } y\} \\
 & r(x, y) \\
= & \{\textit{definition of } r\} \\
 & c(x, y) - f(x, y) \\
\geq & \{\textit{since } c(x, y) \geq 0\} \\
 & -f(x, y) \\
= & \{P1\} \\
 & f(y, x)
\end{array}
$$

Keeping this result in mind, we compute next:

$$
\begin{array}{ll}
 & \langle +x : H.x : e.x \rangle \\
= & \{\textit{definition } e.x\} \\
 & \langle +x : H.x : \langle +z :: f(z, x) \rangle \rangle \\
= & \{\textit{range splitting}\} \\
 & \langle +x : H.x : \langle +z : H.z : f(z, x) \rangle + \langle +z : \neg H.z : f(z, x) \rangle \rangle \\
= & \{\textit{distribution, unnesting}\} \\
 & \langle +x, z : H.x \wedge H.z : f(z, x) \rangle + \langle +x, z : H.x \wedge \neg H.z : f(z, x) \rangle \\
= & \{\textit{by } P1 \textit{ the first term drops out}\} \\
 & \langle +x, z : H.x \wedge \neg H.z : f(z, x) \rangle \\
\leq & \{\textit{by the result kept in mind}\} \\
 & 0
\end{array}
$$

From this, we get

$$
\begin{array}{ll}
 & \langle +x : H.x : e.x \rangle \leq 0 \\
\Rightarrow & \{\textit{since } H.v \textit{ and } e.v > 0\} \\
 & \langle \exists x : H.x : e.x < 0 \rangle \\
\Rightarrow & \{\textit{by } P4\} \\
 & H.s
\end{array}
$$

Hence we arrive at the following important result:

LEMMA 2. $\langle \forall v : sur.v : v \text{ path } s \rangle$.

Using this result, we can now establish the boundedness of the altitude function. First, we observe for all surplus vertices v:

$$
\begin{array}{ll}
 & sur.v \\
\Rightarrow & \{\textit{Lemma } 2\} \\
 & v \text{ path } s \\
\Rightarrow & \{\textit{Lemma } 1\} \\
 & alt.v < alt.s + n
\end{array}
$$

This means that the existence of an upper bound for altitudes depends crucially on the existence of an upper bound for $alt.s$. By P6, we are free to choose such a bound as long as it is at least n. In order to get the least possible bound, we add

the requirement $alt.s \leq n$ to our specification, replacing P6 by the stronger

$$\textbf{invariant } \langle \forall v :: alt.v \geq 0 \rangle \wedge alt.t = 0 \wedge alt.s = n. \tag{P11}$$

Using P11 and our previous observation, we now conlude that $\langle \forall v : sur.v : alt.v < 2n \rangle$. Notice that from P11 we also get $alt.s < 2n$ and $alt.t < 2n$. We now add the additional constraint

$$\textbf{invariant } \langle \forall v : v \in V^- \wedge e.v = 0 : alt.v < 2n \rangle, \tag{P12}$$

and sum up the results above in the following:

THEOREM 3. $\langle \forall v :: 0 \leq alt.v \wedge alt.v < 2n \rangle$.

An immediate consequence of this result is the boundedness of a.

3.6 Refinement 4: Progress Again

This is our final refinement step. After this step, our specification will be in a form that can be directly translated into a program. The properties that need further refinement are P9 and P10, since leads-to properties cannot be directly transformed into program text.

We have to eliminate the existential quantifications in both P9 and P10. We do this by applications of the disjunction rule for $\mapsto$. Doing this for P9 we obtain, omitting the range $sur.v$:

$$\begin{array}{ll} & \text{P9} \\ = & \{definition\} \\ & \langle \exists v :: dis.v \rangle \wedge a = k \mapsto a > k \\ = & \{predicate\ calculus\} \\ & \langle \exists v :: dis.v \wedge a = k \rangle \mapsto a > k \\ \Leftarrow & \{disjunction\ rule\ for \mapsto\} \\ & \langle \forall v :: dis.v \wedge a = k \mapsto a > k \rangle \\ \Leftarrow & \{definition\ of \mapsto\} \\ & \langle \forall v :: dis.v \wedge a = k \textbf{ ensures } a > k \rangle \end{array}$$

So we propose the refined:

$$\langle \forall v : sur.v : dis.v \wedge a = k \textbf{ ensures } a > k \rangle. \tag{P13}$$

We apply the same technique to P10. For brevity's sake, we omit the ranges $sur.v$ and $r(v, w) > 0$.

$$\begin{array}{ll} & \text{P10} \\ = & \{definition\} \\ & \langle \exists v :: \langle \exists w :: alt.v = alt.w + 1 \rangle \rangle \wedge b = l \mapsto b > l \\ = & \{predicate\ calculus\} \\ & \langle \exists v, w :: alt.v = alt.w + 1 \rangle \wedge b = l \mapsto b > l \\ = & \{predicate\ calculus\} \\ & \langle \exists v, w :: alt.v = alt.w + 1 \wedge b = l \rangle \mapsto b > l \\ \Leftarrow & \{disjunction\ rule\} \\ & \langle \forall v, w :: alt.v = alt.w + 1 \wedge b = l \mapsto b > l \rangle \\ \Leftarrow & \{definition\ of \mapsto\} \\ & \langle \forall v, w :: alt.v = alt.w + 1 \wedge b = l \textbf{ ensures } b > l \rangle \end{array}$$

Hence, we propose the refined:

$$\langle \forall v, w : sur.v \wedge r(v, w) > 0 : alt.v = alt.w + 1 \wedge b = l \textbf{ ensures } b > l \rangle. \quad \text{(P14)}$$

3.7 The Complete Refined Specification

The abbreviations we introduced are summarized below:

$$\begin{aligned} e.v &= \langle +x :: f(x, v) \rangle \\ r(v, w) &= c(v, w) - f(v, w) \\ a &= \langle +v :: alt.v \rangle \\ sur.v &\equiv v \in V^- \wedge e.v > 0 \\ dis.v &\equiv \langle \forall w : r(v, w) > 0 : alt.v \leq alt.w \rangle \end{aligned}$$

Our specification consists of the following properties:

(P0) **invariant** $\langle \forall v, w :: f(v, w) \leq c(v, w) \rangle$.
(P1) **invariant** $\langle \forall v, w :: f(v, w) = -f(w, v) \rangle$.
(P4) **invariant** $\langle \forall v : v \neq s : e.v \geq 0 \rangle$.
(P7) **invariant** $\langle \forall v, w : r(v, w) > 0 : alt.v \leq alt.w + 1 \rangle$.
(P8) $a = k$ **unless** $a > k$.
(P11) **invariant** $\langle \forall v :: alt.v \geq 0 \rangle \wedge alt.t = 0 \wedge alt.s = n$.
(P12) **invariant** $\langle \forall v : v \in V^- \wedge e.v = 0 : alt.v < 2n \rangle$.
(P13) $\langle \forall v : sur.v : dis.v \wedge a = k$ **ensures** $a > k \rangle$.
(P14) $\langle \forall v, w : sur.v \wedge r(v, w) > 0 : alt.v = alt.w + 1 \wedge b = l$ **ensures** $b > l \rangle$.

In addition, we require that b be bounded from above. This concludes our refinements.

4. DERIVING THE PROGRAM FROM THE REFINED SPECIFICATION

The final step in our program development consists of writing the program text. First we develop the **always**-section of the program to include our definitions of e and r. The initial conditions are determined by the requirement to establish all invariants initially. Finally, the statements are derived from P13 and P14, after which our program is complete.

4.0 Always-Section

The **always**-section of the program holds the definitions of e and r. This gives us

always
$\langle \Box\, v :: e.v = \langle +x :: f(x, v) \rangle \rangle$
$\Box\, \langle \Box\, v, w :: r(v, w) = c(v, w) - f(v, w) \rangle$

4.1 Initially-Section

We have to give initial values for f (which is mentioned in P0, P1, P4, P7, and P12) and for *alt* (which occurs in P7, P11, and P12). By P11, we have no choice for the altitudes of source and sink: $alt.t = 0$ and $alt.s = n$. But now, in general, P7's validity is in danger for $v = s$ or $w = t$. We would be fine if either s had no outgoing edges or t had no incoming edges in the initial R. But the latter cannot be achieved in general without violating P4. So our only choice is to see

to it that initially

$$\langle \forall w : r(s, w) = 0 \rangle.$$

Solving this equation for f and setting all other values of f to zero, we define initially (using $\sim$ to separate the different cases):

$$\begin{array}{lrll} \langle \Box\ v, w :: f(v, w) = & 0 & \textbf{if} & v \neq s \wedge w \neq s \\ & c(v, w) & \textbf{if} & v = s \qquad \sim \\ & -c(v, w) & \textbf{if} & w = s \qquad \rangle \end{array}$$

Note that the last line is needed to establish P1 initially. Furthermore, note that f is well defined, since $c(s, s) = 0$ by the definition of c.

If we now instantiate $w := t$ in P7, we get for all internal vertices x in general, the requirement $alt.x \leq 1$. Since we want to increase altitudes after all, we choose initially:

$$\langle \Box\ v : v \in V^- : alt.v = 1 \rangle \ \Box\ alt.s = n \ \Box\ alt.t = 0.$$

4.2 Assign-Section

Next we derive two sets of statements from the two ensures properties of our final specification.

Statements Derived from P13

P13 serves to ensure that a is incremented eventually. The simplest way to achieve this is to increment one altitude at a time. Let us now consider which new value for the altitude is appropriate. Towards this end, we define the expression $M.v$ as follows:

$$M.v = \langle \min\ w : r(v, w) > 0 : alt.w \rangle.$$

Note that for a surplus vertex v, the range of the above min in nonempty, since by Lemma 2, $sur.v$ implies v path s; that is, in particular, the existence of some vertex w with $r(v, w) > 0$.

The new value of the altitude has to be such that P7 is maintained. So this new value can be at most one greater than $M.v$. On the other hand, it has to be at least one greater than $M.v$ to be guaranteed to increase at all. Hence, the only possible choice is

$$alt.v := M.v + 1.$$

The first set of statements of the program is now obtained by observing that

$$\begin{array}{ll} & dis.v \\ = & \{definition\ dis\} \\ & \langle \forall w : r(v, w) > 0 : alt.v \leq alt.w \rangle \\ = & \{arithmetic\} \\ & alt.v \leq \langle \min\ w : r(v, w) > 0 : alt.w \rangle \\ = & \{definition\ M\} \\ & alt.v \leq M.v \end{array}$$

Gathering this last term and the range of P13 into the conditional part, we get, for the first set of statements:

$$\langle \Box\, v :: alt.v := M.v + 1 \textbf{ if } sur.v \wedge alt.v \leq M.v \rangle. \qquad \text{(T0)}$$

We also add the definition of M to the **always**-section of the program.

That each statement in T0 indeed preserves P0, P1, P4, P7, P8, P11, P12, and the **unless** part of P13 and P14 is proved purely mechanically, and is therefore omitted; the existential part of P13 holds by construction of T0.

Statements Derived from P14

Our refinement process with respect to P14 is aimed at guaranteeing that net flow will be transferred from one vertex to another under invariance of P1. Translating P14 directly into this goal, adding the range of P14 to the conditional part, we obtain

$$\langle \Box\, v, w :: f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w) \textbf{ if } sur.v \wedge r(v, w) > 0 \wedge alt.v = alt.w + 1 \rangle, \qquad \text{(T1)}$$

with some positive d that is developed next. First of all, we want d to be as large as possible. How big can d be at most? Since we need to maintain P0, $d(v, w) \leq r(v, w)$. Similarly, by P4, $d \leq e.v$. So to maximize d, we set

$$d(v, w) = e.v \text{ min } r(v, w),$$

which is positive, since both $e.v$ and $r(v, w)$ are. We add this definition to the **always**-section.

The proofs of all properties except P14 are straightforward, and therefore omitted here. What is left is finding a suitable choice for b and proving that T1 satisfies P14. P14 being the **ensures** property established by some statement in T1, we need to prove two things: the **unless** part and the existential part. Let us first concentrate on the existential part.

Since T1 is a set of conditional assignments, by D0 of Section 1.2, we can write the proof obligation for given v and w as two separate requirements. After some straightforward simplification, we then get for all v, w, for which $sur.v \wedge r(v, w) > 0$:

$$alt.v = alt.w + 1 \wedge b = l \Rightarrow \mathbf{wp}(f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w), b > l)$$

and

$$false \Rightarrow b > l.$$

We note that the second proof obligation is trivially satisfied.

In Section 3.4, b was introduced to capture progress made by vanishing net flows and disappearing edges of the residual graph. The only net flows affected by some fixed statement in T1 are $e.v$ and $e.w$, and the only edges affected are $r(v, w)$ and $r(w, v)$. Let us therefore investigate the preconditions under which

net flows vanish and edges of the residual graph disappear:

$$
\begin{aligned}
&\mathbf{wp}(f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w), e.v = 0)\\
&= \{\textit{definition } \mathbf{wp}, e.v \textit{ has exactly one occurrence of } f(w, v)\}\\
&\quad e.v - d(v, w) = 0\\
&= \{\textit{definition } d \textit{ and arithmetic}\}\\
&\quad e.v = e.v \min r(v, w)\\
&= \{\textit{property of min}\}\\
&\quad e.v \leq r(v, w)
\end{aligned}
\tag{C0}
$$

$$
\begin{aligned}
&\mathbf{wp}(f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w), e.w = 0)\\
&= \{\textit{definition } \mathbf{wp}, e.w \textit{ has exactly one occurrence of } f(v, w)\}\\
&\quad e.w + d(v, w) = 0\\
&= \{\textit{since } e.w \geq 0, d(v, w) > 0\}\\
&\quad \textit{false}
\end{aligned}
\tag{C1}
$$

$$
\begin{aligned}
&\mathbf{wp}(f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w),\\
&\qquad r(v, w) = 0)\\
&= \{\textit{definition } \mathbf{wp} \textit{ and } r(v, w)\}\\
&\quad r(v, w) - d(v, w) = 0\\
&= \{\textit{arithmetic}\}\\
&\quad r(v, w) = e.v \min r(v, w)\\
&= \{\textit{property of min}\}\\
&\quad r(v, w) \leq e.v
\end{aligned}
\tag{C2}
$$

$$
\begin{aligned}
&\mathbf{wp}(f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) -\\
&\qquad d(v, w), r(w, v) = 0)\\
&= \{\textit{definition } \mathbf{wp} \textit{ and } r(w, v)\}\\
&\quad r(w, v) + d(v, w) = 0\\
&= \{\textit{since } r(w, v) \geq 0, d(v, w) > 0\}\\
&\quad \textit{false}
\end{aligned}
\tag{C3}
$$

We now consider two types of expressions based on these four possibilities. Our goal is to find an expression that is increased by the assignment in all cases.

The first type is of the form

$$\langle alt.v \textbf{ if } e.v = 0\rangle + \langle alt.w \textbf{ if } e.w = 0\rangle.$$

For its value *pre*0 in a state satisfying the precondition of the assignment, we get, since $e.v > 0$ and $\langle alt.w \textbf{ if } e.w > 0\rangle \leq alt.w$,

$$pre0 \leq alt.w.$$

Its value in a state satisfying the postcondition of the assignment is, using C0 and C1:

$$post0 = \langle alt.v \textbf{ if } e.v \leq r(v, w)\rangle.$$

The second type of expression is

$$\langle alt.v \textbf{ if } r(v, w) = 0 \wedge alt.v = alt.w + 1\rangle + \langle alt.w \textbf{ if } r(w, v) = 0 \wedge alt.w = alt.v + 1\rangle.$$

Using C2 and the fact that $alt.v = alt.w + 1$ holds in both pre- and postconditions, we get the following values:

$$\begin{aligned} pre1 &= 0 \\ post1 &= \langle alt.v \textbf{ if } r(v, w) \leq e.v \rangle. \end{aligned}$$

Now we add up corresponding values for pre- and postconditions:

$$\begin{aligned} & post0 + post1 \\ = & \{substitute\} \\ & \langle alt.v \textbf{ if } e.v \leq r(v, w) \rangle + \langle alt.v \textbf{ if } r(v, w) \leq e.v \rangle \\ \geq & \{arithmetic\} \\ & alt.v \\ > & \{since\ alt.v = alt.w + 1\} \\ & alt.w \\ \geq & \{substitute\} \\ & pre0 + pre1 \end{aligned}$$

Together with the fact that all other values of e and r besides the ones mentioned are invariant under the assignment, this argument establishes that b satisfying

$$b = \langle +y : e.y = 0 : alt.y \rangle + \langle +y, z : r(y, z) = 0 \wedge alt.y = alt.z + 1 : alt.y \rangle$$

increases when the execution of some statement in T1 is effective, i.e., that the existential part of P14 is satisfied. Also, by Theorem 3, b is bounded from above.

It turns out, however, that the **unless** part of P14 with respect to the statements in T0 is not satisfied (the reader is encouraged to work out a counterexample). But since a is increased by any effective statement in T0 and is unchanged by the statements in T1, $b' = (a, b)$ with lexicographic ordering does the trick, instead of b. For one, b' is bounded from above since both a and b are. In addition, T1 still satisfies the existential part of P14. Furthermore, it is straightforward to show that all statements in T0 and T1 satisfy the **unless** part of P14, so all requirements have been met.

4.3 The Complete Program

Below is the complete program that we have derived.

program *Maximum Flow*
initially
$\quad \langle \Box v, w :: f(v, w) = \begin{array}{rll} 0 & \textbf{if} & v \neq s \wedge w \neq s \sim \\ c(v, w) & \textbf{if} & v = s \quad\quad \sim \\ -c(v, w) & \textbf{if} & w = s \quad\quad \rangle \end{array}$
$\quad \Box \langle \Box v : v \in V^{-} : alt.v = 1 \rangle \Box alt.s = n \Box alt.t = 0$
always
$\quad \langle \Box v, w :: r(v, w) = c(v, w) - f(v, w) \rangle$
$\quad \Box \langle \Box v :: e.v = \langle +x :: f(x, v) \rangle \rangle$
$\quad \Box \langle \Box v :: M.v = \langle \min w : r(v, w) > 0 : alt.w \rangle \rangle$
$\quad \Box \langle \Box v, w :: d(v, w) = e.v \min r(v, w) \rangle$
assign
$\quad \langle \Box v :: alt.v := M.v + 1 \textbf{ if } sur.v \wedge alt.v \leq M.v \rangle$
$\quad \Box \langle \Box v, w :: f(v, w), f(w, v) := f(v, w) + d(v, w), f(w, v) - d(v, w)$
$\quad\quad \textbf{if } sur.v \wedge r(v, w) > 0 \wedge alt.v = alt.w + 1 \rangle$
end

5. DISCUSSION

The program we obtained exhibits a high degree of nondeterminism. It can be further refined to exploit the characteristics of specific architectures. Below we give a sketch of what such further refinements would look like.

For a sequential architecture, the refinement has to deal with scheduling the execution of statements and efficiently evaluating their conditions to avoid expensive recomputations. Work in this direction is described in detail in [5]. The sequential algorithm obtained there has a time complexity of $O(nm \log(n^2/m))$, where m is the number of edges with positive capacity.

Refinement for a distributed architecture consists of mapping statements to processes and of replacing global data structures by local ones. Again, [5] contains an extensive discussion on how to implement the algorithm efficiently on both synchronous and asynchronous architectures, achieving a time bound of $O(n^2 \log n)$ using $O(n)$ processors and $O(m)$ local storage.

The derivation we presented is instructive in several respects. An algorithm that, when presented in conjunction with the usual a-posteriori proof, might strike the reader as a miraculous invention can be derived with top-down design by a sequence of more or less consequential refinement steps. This is not to say that there is not a great deal of creativity involved in this process. But, in this way of design, the inventive steps are more easily distinguished from results that have been obtained by mere calculations.

The main problem in this kind of development is to decide at each step which property to refine next. Even though there are a number of useful heuristics (such as identifying formulas that cannot be easily manipulated syntactically), more research is needed to identify additional criteria for refinement.

In our example, there were two main inventions: one was the idea of a preflow, while the other was the introduction of altitudes, together with invariant P7. Most of the other steps were suggested by the syntactic shape of our formulas and the UNITY formalism. Furthermore, the derivation of the actual program text was a straightforward task, once our refinements had progressed down to the level of **ensures** properties.

The advantage of this approach is that all the reasoning about correctness is done in the domain of logic rather than in the domain of program executions. Experience has shown that proving parallel programs by looking at their possible executions is cumbersome and error prone, and therefore had best be avoided. The UNITY view of a program is that of a mathematical object which is the result of a series of stepwise refinements of specifications and which, in a final step, can be mapped to a variety of different target architectures.

UNITY is the subject of much ongoing research with many questions still waiting to be answered. Our experience so far suggests that it is a powerful tool in all stages of the design of concurrent programs.

ACKNOWLEDGMENTS

We are indebted to Jay Misra for his continued help and encouragement. We are also grateful to the members of the Austin Tuesday Afternoon Club and the Eindhoven Tuesday Afternoon Club for their numerous suggestions. Comments

from Wim Feijen helped to improve the presentation of the paper. C. S. Scholten provided valuable insights on an earlier draft of the article. Thanks are also due to the referees who provided valuable criticisms of this paper.

REFERENCES

0. BOLLOBÁS, B. *Graph Theory, An Introductory Course.* Springer Verlag, New York, 1979.
1. CHANDY, K. M., AND MISRA, J. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Mass., 1988.
2. DIJKSTRA, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood, Cliffs, N.J., 1976.
3. DIJKSTRA, E. W. Our proof format. EWD 999, Univ. of Texas at Austin, Jan. 1987.
4. FORD, L. R., AND FULKERSON, D. R. *Flows in Networks.* Princeton University Press, Princeton, N.J., 1962.
5. GOLDBERG, A. V. Efficient graph algorithms for sequential and parallel computers. Ph.D. dissertation, MIT, Feb. 1987. MIT/LCS/TR-374.
6. GOLDBERG, A. V., TARJAN, R. E. A new approach to the maximum flow problem. In *Proceedings 18th ACM Symposium on Theory of Computing.* ACM, New York, 1986, 136–146.
7. GRIES, D. *The Science of Programming.* Springer-Verlag, New York, 1981.
8. KARZANOV, A. V. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.* (1974), 434–437.
9. LYNCH, N. A., AND TUTTLE, M. R. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing* (Vancouver, B.C., 1987). ACM, New York, 1987.
10. MANNA, Z., AND PNUELI, A. How to cook a temporal proof system for your pet language. In *Proceedings 10th Annual ACM Symposium on Principles of Programming Languages* (Austin, Tex. 1983). ACM, New York, 1983.
11. OWICKI, S., AND GRIES, D. An axiomatic proof technique for parallel programs. *Acta Inf. 6*, 1 (1976), 319–340.
12. OWICKI, S., AND LAMPORT, L. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst. 4*, 3 (July 1982), 445–495.
13. STASKAUSKAS, M. The formal specification and design of a distributed electronic funds transfer system. *IEEE Trans. Comput. 37*, 12 (Dec. 1988), 1515–1528.

Received August 1988; revised August 1989; accepted September 1989