Transformational Design and Implementation Of A New Efficient Solution To The Ready Simulation Problem*

Bard Bloom†
Department of Computer Science, Cornell University
Ithaca, New York, USA
bard@cs.cornell.edu

Robert Paige‡
Department of Computer Science, New York University
New York, New York, USA
paige@cs.nyu.edu

January 10, 1995

Abstract

A transformational methodology is described for simultaneously designing algorithms and developing programs. The methodology makes use of three transformational tools - dominated convergence, finite differencing, and real-time simulation of a set machine on a RAM. We illustrate the methodology to design a new $O(mn + n^2)$-time algorithm for deciding when n-state, m-transition processes are ready similar, which is a substantial improvement on the $\Theta(mn^6)$ algorithm presented in [6]. The methodology is also used to derive a program whose performance, we believe, is competitive with the most efficient hand-crafted implementation of our algorithm. Ready simulation is the finest fully abstract notion of process equivalence in the CCS setting.

1 Introduction

Currently there is a wide gap between the goals and practices of research in the theory of algorithm design and the science of programming, which we believe is

*A preliminary version of this paper appeared in the Conference Record of the First North American Process Algebra Workshop.

†Supported by NSF grants CCR-9003441 and CCR-9223183.

‡Supported by ONR grant N00014-93-1-0924, AFOSR grant AFOSR-91-0308, and NSF grant MIP-9300210

detrimental to both areas. We seek to bridge this gap so that new algorithms and algorithmic principles can be effectively programmed, and that broadly applicable algebraic methods and a syntax-driven approach can make algorithm design easier. This paper illustrates by example how to simultaneously design algorithms and develop programs by transformation, thereby combining the more theoretical aims of the algorithm designer with the more pragmatic concerns of the implementor.

Algorithm designers and programmers have moderately different goals, and this has led to an excessively wide gap between their results. Algorithm researchers manipulate generic design principles that are independent of programming language but strongly dependent on a computational model. They emphasize correctness and complexity at the expense of implementation details. Theoretical results in algorithms are frequently built on top of each other, using reductions from one problem to another, and reusing complexity results without regard for the actual implementation. The only formal system supporting this work is asymptotic notation. This sort of reuse makes complexity-based reducibilities very important. When reducibilities are not used, the overall emphasis on complexity requires that the discussion of new algorithms be highly operational with special emphasis on carefully counting computation steps.

This kind of abstraction from implementation details and occasionally eccentric use of other algorithms is one of the main strengths of algorithms design. Unfortunately, all of this makes a naive literal rendering of an algorithm description into an implementation a risky business strewn with impractical interfaces, bizarre interoperability, and exorbitant numbers of redundant passes over large data structures. A mere constant multiplicative factor or low-order term to a theoretician may give rise to a completely impractical implementation. Thus, an algorithmic result rarely gives rise to more than a hope that a reasonably efficient algorithm exists. It is up to the programmer to find it, sometimes by a great deal of hard work.

The primary goal of the science of programming is to develop methods for producing programs that run correctly on machines. Implementation details are as important as correctness and efficiency. Programs are best developed in explicit notational systems, which support reasoning about all matters of concern. Thus, the development of a formal calculus in which to derive programs is a theoretical and practical goal. To be practical it must offer an improvement over conventional ad hoc approaches. Calculi that ignore basic principles of algorithm and data structure design (*e.g.*, the classical fold/unfold formalism) are likely to end up with long and tedious transformational derivations. By ignoring the complexity considerations that guide algorithm designers and system implementors, such calculi are hard to use, and are likely to produce inefficient programs. Finally, such methods consist mainly of low-level transformations and minute details. This does not help with the ad hocery of program design; indeed, it simply burdens the ad hoc programmer with the need to provide a proof for each low level step. While algorithm theory offers the programmer

hope for a practical implementation, few results in the science of programming offer even that much to algorithm designers.

It is vital to bridge this gap between algorithms and programming and to make results from one community readily available to the other. Algorithmic results are especially important to the pragmatic computer scientist. In part, this is because new more difficult problems that require algorithmic solutions become natural challenges as the easier data processing applications get solved. The formidable problem of producing software tools, such as optimizing compilers for parallel architectures, is also likely to be much harder and more algorithmic than for sequential machines. Because the performance of an implementation on a parallel machine can have a much wider range than on a sequential one (simply because the degenerate form of a concurrent machine is nothing more than a sequential machine), it is likely that the ingenuity of algorithms and their implementations will be more important in that setting than in the sequential milieu.

In this paper we present a transformational methodology based on a synthesis of ideas drawn from the theoretical algorithms and the programming methodology communities. The methodology makes use of an explicit notation to describe and reason about algorithms. This notation together with a small number of powerful transformational tools can be used to design algorithms and derive programs simultaneously. They enhance reliability. These tools, when applied manually as part of a systematic programming methodology, make it harder to introduce bugs, because so many design decisions and implementation details follow from formal 'calculation'. The ordered application of these tools not only provide a proof (i.e., a program derivation), but they help document a reasonable explanation of why and how the algorithm/program works. Analysis is integrated with design and is supported by syntactic and algebraic reasoning without great overhead. All of this naturally facilitates algorithm/program adaptation, enhancement, and general maintenance.

Three basic transformations form the core of our methodology. These transformations capture broadly applicable data structure and algorithm design principles, and also facilitate algorithm verification and analysis. They are,

1. *dominated convergence* [11], a methodology (analogous to dominated convergence in real analysis) for computing least and greatest fixed points of computable functions by generating sequences more efficiently than Tarski sequences [44,19],

2. *finite differencing* [38], a methodology (analogous to polynomial tabulation by difference polynomials) for implementing costly repeated calculations by more efficient incremental ones, e.g. implementing a search through a set for a property using dynamic indexes that support efficient navigation, and

3. *real-time simulation*[35,37] of a set machine on a RAM, a methodology

for providing an efficient physical structure to implement each primitive set-oriented operation (e.g. membership testing $x \in s$) using a constant number of unit-time RAM operations[1].

Three kinds of 'gedanken' experiments have been used to test the viability of these transformational tools. The first kind of experiment tests whether the methodology allows us to reason about complexity of programs based on syntactic considerations. In rough terms we want the cost of executing program constructs to be transparent so that we can reason about analysis algebraically and be guided by such analysis. Syntactic characterizations of amortized and worst case complexities were used to design a hierarchy of complexity-bound specification languages. For each polynomial degree i, we defined a high level functional language L_i whose compiled queries have worst case asymptotic time and space bounded by an i-th degree polynomial in their I/O space[10,12]. These languages are software generators that extend the program generator idea of YACC [29].

The second kind of experiment tests pedagogical efficiency; i.e., whether our methodology helps to describe and analyze complex algorithms more easily than standard algorithm texts. We believe the success of this kind of experiment depends strongly on the success of the first kind. For example, in an undergraduate algorithms course using Aho, Hopcroft, and Ullman's algorithm text[2], we were able to apply our three basic transformations to cover more ground in less time, to shift the emphasis from data structure rendering (as in [2]) to principles of data structure design, and to achieve a greater level of student understanding. This methodology was used to construct a perspicuous derivation of a new solution to the one-symbol DFA minimization problem.

The third kind of experiment tests whether the methodology facilitates the discovery of new algorithms. We believe that the success of this experiment hinges on the success of the first two experiments. Earlier forms of our methodology partly contributed to the discovery of improved solutions to DFA minimization[40,11], regular expression search[15], and bottom-up multi-pattern tree matching[13].

A substantially narrower form of the transformational methodology presented here was used by Paige and Henglein[37] to develop a linear time program to solve the relational attribute closure problem using only list processing operations. An earlier linear time solution due to Beeri and Bernstein [4] relied on arrays. The current paper shows for the first time how our methodology, broadened considerably to derive array-based algorithms, can be used to invent every aspect of a complex algorithm that runs five orders of magnitude faster than the best previous solution. An actual program that implements this algorithm is developed at the same time.

2 The Ready Simulation Problem

2.1 Motivation from Process Algebra

In process algebras such as Milner's CCS [33], a process P is an entity capable of repeatedly participating in uninterpreted atomic actions a. In practice, events such as "the y key on the keyboard is pressed," "the processor sends a message on channel 83," and "circuit component 412 experiences a signal on its input wire" are modeled as atomic actions. The basic operational notion in such languages is $P \xrightarrow{a} P'$, indicating that P is capable of performing action a and thereafter behaving like P'. In general, processes are nondeterministic; P may have several alternative possible behaviors after performing a: that is, $P \xrightarrow{a} P'$ for several P''s.

One of the basic problems in this area is to find good notions of *process equivalence*. For example, a process that simply performs an action a and then stops ought to be the same as one that has several different ways of doing a's and then stopping, since one a or stopped process is the same as another. A wide variety of notions of equivalence have been proposed, *e.g.*[28,31,20,5,21,27,41], appropriate for different kinds of process algebras and conceptual settings.

Process equivalences can be partially ordered by *fineness*: finer notions make more distinctions between processes; coarser ones consider more processes identical. Both fine and coarse notions are useful in theory and practice. For theory, fine notions give more detailed insight into the precise structure of concurrency; coarse notions can give insight into the nature of a particular programming language. In practice, fine notions allow one to prove extremely strong theorems about processes: *e.g.*, prove that a process meets its specification extremely precisely and will behave properly in all reasonable (and, generally, most unreasonable) environments. Coarse notions can only prove weaker theorems — *e.g.*, that the program is correct in composition with other programs in the same language — but such theorems are usually good enough, and (as they are weaker) are more likely to hold. A notion of equivalence is best when it is neither too fine nor too coarse; that is, when it is *fully abstract* for the language in question: when two processes are semantically equivalent if and only if they are indistinguishable in the language: *e.g.*, iff they produce the same sequences of actions in all contexts in the language (*i.e.*, they produce visibly the same results). In general, coarse notions are introduced because they are fully abstract for some language of interest.

In this study, we consider two of the finest (most discriminating) notions: ready simulation and (strong) bisimulation. Bisimulation [32,33,3,5] is generally regarded as the finest usable notion of process equivalence in this setting. If processes P and Q are bisimilar, written $P \leftrightarrows Q$, then at all future times, they have exactly the same set of nondeterministic choices available.

Bisimulation is a meaningful notion in many settings beyond process algebra (*e.g.*, [34,30]). It admits several powerful proof methods, and protocol-

verification environments based on bisimulation have been built [16]. Furthermore, bisimulation of n-state, m-transition processes can be computed in $O(m \lg n)$ time [39], making verification of even relatively large protocols a viable possibility.[1] Such a possibility has been realized by Fernandez, who extended the algorithm found in [39] to work with more than one relation in order to solve the full bisimulation problem. Fernandez also used fixed point methods to form the initial portion of a derivation of his algorithm. He developed an implementation as part of the Aldébaran verification tool[24].

However, bisimulation is too fine in general. Bisimulation pays great attention to exactly *when* decisions were made. For example, consider a *lossy queue*: a communication channel that accepts messages on one end, and nondeterministically either delivers them at the other end or loses them. (While this is not a data structure many programs are likely to build intentionally, it is a respectable approximation of many physical communication mechanisms, and thus is a reasonable thing to appear in protocol analyses.) There are several versions of the lossy queue. If queue Q_1 can only lose messages when they are enqueued, and queue Q_2 can lose messages on enqueueing or dequeueing, then Q_1 and Q_2 are not bisimilar. However, they behave identically in all CCS programs. Thus it seems reasonable to wish to consider them the same. Ready simulation is a slight coarsening of bisimulation that does identify them, and, more generally, it ignores irrelevant differences in decision times.

In [6,7,8] we formalize the concept of a "CCS-like language" as a GSOS language; that is, one whose rules are defined in a style that generalizes CCS's style. Using this definition, we show that bisimulation is not fully abstract for any CCS-like language; indeed, in the lossy queue example Q_1 and Q_2 are indistinguishable in all CCS-like languages.

Based on that analysis, we proposed the notion of *ready simulation*, which is the finest notion of process equivalence that is fully abstract for *some* CCS-like language. That is,

1. if two processes are ready similar, then one may be replaced by the other freely in any program written in a CCS-like language; and

2. if two processes are not ready similar, then there is reason to consider them different; e.g. a CCS-like language and a program in it in which the two will produce visibly different results.

See [7,27,30]. for more details. Ready simulation is thus more appropriate for CCS-like languages for foundational reasons. The theory and practice of bisimulation are well-developed.

[1] Note that n is the number of states in the process, rather than the size of the process. Even for restricted calculi, state spaces may be exponentially larger than code. This is a fundamental weakness of all state-space exploration methods.

2.2 Problem Definition

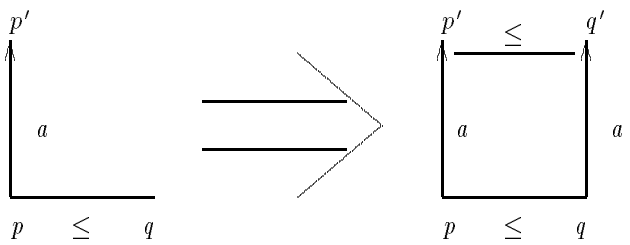
The mathematics of ready simulation is straightforward. Let A be the (finite) alphabet of action symbols. Processes are labelled transition systems $P = \left\langle \Sigma_P, \rightarrow_P, i_P \right\rangle$, where Σ_P is a finite set of states, $i_P \in \Sigma_P$ is the *initial state* and $\rightarrow_P$ is a *transition relation*, a subset of $\Sigma_P \times A \times \Sigma_P$. Lower-case letters range over states. Transitions are written in infix: $p \xrightarrow{a} p'$. We omit the subscript on the transition relation $\xrightarrow{a}$ for both processes' transition relation, and write $p \xrightarrow{a}$ iff $\exists p'.p \xrightarrow{a} p'$. We write $p \xrightarrow{a} p' \in P$ iff $p \xrightarrow{a} p'$ is a labelled transition of process P.

Let P and Q be finite-state processes. A relation $\leq$ that is a subset of $\Sigma_P \times \Sigma_Q$ is a *ready simulation relation* if, whenever $p \in \Sigma_P$, $q \in \Sigma_Q$, and $p \leq q$, the following two conditions hold:

1. For every p' and a such that $p \xrightarrow{a} p'$, there is some q' such that

$$q \xrightarrow{a} q' \text{ and } p' \leq q'$$

For example, the empty relation is (vacuously) a ready simulation relation.



2. p and q have precisely the same set of initial actions; i.e., for all a,

$$p \xrightarrow{a} \text{ iff } q \xrightarrow{a}. \tag{1}$$

State p is said to be ready similar to state q if $p \leq q$. Process P is a ready simulation approximation of process Q, for which we write $P \sqsubseteq Q$, if there exists a ready simulation relation $\leq$ where $i_P \leq i_Q$; P and Q are ready similar, $P \eqsim Q$, if $P \sqsubseteq Q \sqsubseteq P$.

If $\leq_1$ and $\leq_2$ are two ready simulation relations between finite state processes P and Q, then it is easy to see that relation $\leq$, defined by the rule $p \leq q$ iff $p \leq_1 q$ or $p \leq_2 q$, is also a ready simulation relation.[2] It follows that there is a

[2]Indeed, the union of any set of ready simulation relations between arbitrary processes is a ready simulation relation.

unique maximum ready simulation relation, denoted by $\leq_{P,Q}$, between P and Q, which can be used to determine whether P is a ready simulation approximation of Q; that is, $i_P \leq_{P,Q} i_Q$ iff $P \sqsubseteq Q$.

Figure 1 gives examples of maximum ready simulation relations $\leq_{P,Q}$ and $\leq_{Q,P}$ between processes P and Q. Since the starting states are ready similar with respect to both of these relations (i.e., $i_P \leq_{P,Q} i_Q$ and $i_Q \leq_{Q,P} i_P$), we see that P and Q are ready similar.

Note that the ready simulation relations $\leq_{P,Q}$ and $\leq_{Q,P}$ shown in Figure 1 are not inverses. The apparent asymmetry of ready simulation relations can be contrasted with the symmetry of bisimulation. If P and Q are two finite state processes, then a relation $\leq$ that is a subset of $\Sigma_P \times \Sigma_Q$ is a *bisimulation relation* if, whenever $p \in \Sigma_P$, $q \in \Sigma_Q$, and $p \leq q$, the following two conditions hold:

1. For every p' and a such that $p \xrightarrow{a} p'$, there is some q' such that

$$q \xrightarrow{a} q' \text{ and } p' \leq q'$$

2. For every q' and a such that $q \xrightarrow{a} q'$, there is some p' such that

$$p \xrightarrow{a} p' \text{ and } p' \leq q'$$

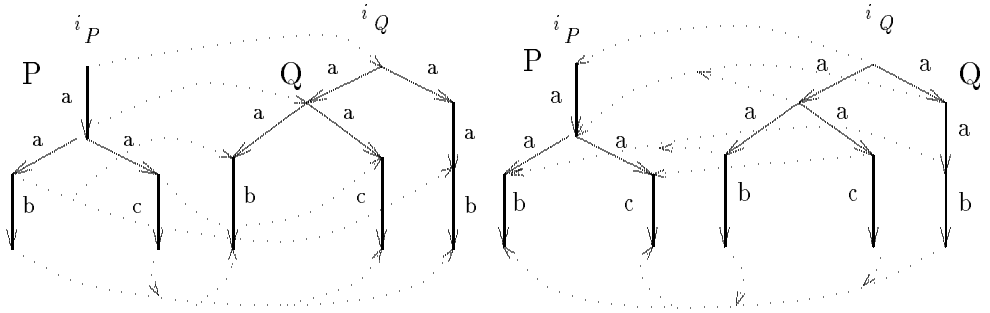
As in the case of ready simulation there exist unique maximum bisimulation relations $\leq_{P,Q}$ and $\leq_{Q,P}$. But unlike ready simulation, these bisimulation relations are inverses. Because of this symmetry, processes P and Q are bisimular ($P \leftrightarrows Q$) iff $i_P \leq_{P,Q} i_Q$. It is straightforward to show that if P and Q are bisimilar, then they are ready similar. However, the converse does not hold. Figure 1 gives an example of two processes that are ready similar but not bisimilar.

This study expands the practical side of the theory of ready simulation by giving an algorithm that computes ready simulations in time $O(n^2+nm)$, where $n = \max(|\Sigma_P|, |\Sigma_Q|)$, and m is the total number of transitions in P and Q. The previous best time was $\Theta(mn^6)$ presented in [6]. An $O(mn^2)$ algorithm for a related problem is presented in [17]. Throughout this paper we assume a sequential RAM model of computation under a uniform cost criterion[1].

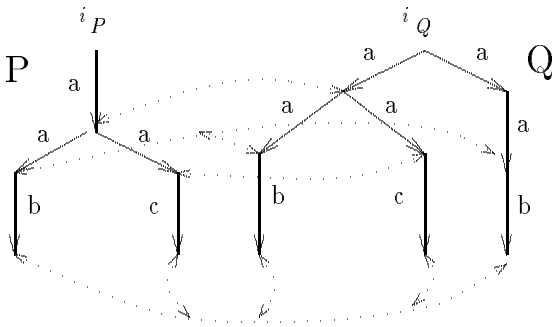
3 Algorithm Derivation

3.1 Methodology

We will present the new algorithm as the end-product of a semi-formal derivation. (All the main steps of a fully formal derivation are given with informal proofs provided to 'justify' the more difficult steps.) The methodology underlying this derivation integrates formal specification, algorithm design, proof, and



a. Dotted arrows represent maximum ready simulation relations $\leq_{P,Q}$ and $\leq_{Q,P}$.



b. Dotted double arrows represent maximum bisimulation relations between P and Q.

Figure 1: Maximum Ready Simulation and Bisimulation Relations Between Processes P and Q.

analysis within a unified framework. We begin by presenting a formal specification of the function to be computed, and map this function by successive meaning preserving program transformations into an efficient implemention. Each transformation being selected is either an obvious simplification or is guided by complexity considerations. Surprisingly, the step-by-step description below corresponds closely to the systematic way in which the new algorithm was discovered. In using this syntax-directed approach no inspired decisions (called 'eureka' steps by Burstall and Darlington[9]) were needed.

3.2 Derivational Tools

Before proceeding with the derivation, we first need to present a few definitions, formal tools, and convenient notations. A partially ordered set $(X, \leq)$ is a set X with a reflexive, anti-symmetric, transitive relation $\leq$. It has the *descending*

chain condition if every totally ordered subset of X that includes a maximum element is finite. Given a function $f : X \rightarrow X$, we define $f^0(x) = x$ and $f^{n+1}(x) = f(f^n(x))$, and use the term $\mathrm{gfp}(x = f(x))$ to denote the greatest solution to the equation $x = f(x)$, *viz.* the greatest fixed point of f.

We will use the fixed point theorem and dominated convergence argument just below as tools for proving correctness of a formal specification and taking the initial step in the algorithm derivation.

Definition 3.1 *A semilattice is a structure $(L, \sqcap, 1)$, where L is a set, $\sqcap$ is a binary meet (i.e., greatest lower bound) operation, and 1 is the greatest element in the partial order induced by $\sqcap$: $x \leq y$ iff $x = x \sqcap y$.*

Theorem 3.2 (Tarski)*[44]: Given a semilattice $(L, \sqcap, 1)$ which has the descending chain condition, and given a monotone function $f : L \rightarrow L$, then*

$$(\max Y | Y \leq f(Y)) = \mathrm{gfp}(Y = f(Y)) = f^i(1)$$

for some finite i.

Theorem 3.3 (Dominated Convergence - [11])*: Under the same conditions as in Theorem 3.2, and given any sequence s_i such that*

1. $s_0 = 1$
2. $s_{i+1} = s_i$ *if* $s_i = f(s_i)$
3. $s_i > s_{i+1} \geq f(s_i)$ *otherwise.*

then s_i converges to $s_k = \mathrm{gfp}(Y = f(Y))$ for some finite k.

Dominated convergence can be used to compute fixed points over a variety of lattice-theoretically defined datatypes. Partition and set lattices are obvious examples. To solve the ready simulation problem, we will compute greatest fixed points of functions defined on finite sets of pairs, according to the following Corollary.

Corollary 3.4 *If S is a finite set, $L = \wp(S)$ is the powerset of S, $\sqcap$ is intersection, then* $\mathrm{gfp}(Y = f(Y))$ *can be computed in $O(|S|)$ iterations by the following code:*

```
Y := S;
while ∃ x ∈ (Y − f(Y)) loop
    % execution frequency O(|S|)
    Y := Y - {x}
end loop
```

Proof: It is well-known that $(L, \cap, S)$ is a semilattice with set inclusion $\subseteq$ as the partial order. The code assigns the values s_i to Y in succession, where $s_0 = S$, $s_{i+1} = s_i$ if $s_i = f(s_i)$, and $s_{i+1} \in \{s_i - \{x\} : x \in s_i - f(s_i)\}$ otherwise. It is clear by induction on i that $f^{i+1}(S) \subseteq f(s_i) \subseteq s_{i+1} \subseteq s_i \subseteq S$ for all i. The corollary is thus a simple consequence of the dominated convergence theorem. □

Whereas dominated convergence computes fixed points according to a rudimentary strategy, the finite differencing transformation [38] maintains and exploits program invariants that are used to implement this strategy efficiently. The method has been called 'strengthening invariants' by Dijkstra[22] and Gries[26]. Related ideas go back to the compiler optmization method of strength reduction[18], iterator inversion[23], and set theoretic strength reduction[25]. The essential idea is to avoid costly repeated calculations $f(x_1, ..., x_k)$ occurring in a program region R by maintaining equality invariants of the form

$$C = f(x_1, ..., x_k) \tag{2}$$

where $C, x_1, ..., x_k$ are distinct variables.

The finite difference transformation inserts code on entry to R that establishes invariant (2). Within R just before each assignment $x_i := g(x_i)$ to a variable on which f depends spoils invariant (2), code is inserted to update variable C in order to re-establish invariant (2) immediately after the assignment to x_i. All calculations of $f(x_1, ..., x_k)$ that are made redundant by these steps are replaced by the variable C. Finite differencing improves performance when the cumulative cost of repeatedly computing $f(x_1, ..., x_k)$ in the unoptimized program exceeds the cost of establishing and maintaining invariant (2) in the optimized program.

Finite differencing rules can be formalized using Hoare Logic.

Definition 3.5 *Let S be a command in an imperative programming language, and let φ and φ' be assertions in a formal language, where the free variables of φ and φ' are also program variables. The Hoare formula $\{\varphi\}S\{\varphi'\}$ is said to be satisfied iff every terminating execution of S that satisfies φ in its initial state satisfies φ' in its final state.*

The following Lemma, which is stated without proof, provides a basic schema that is used throughout this paper for designing correct code to maintain equality invariants of the form (2). This code is made more efficient by straightforward simplification.

Lemma 3.6 *(**Finite Difference Schema**) Let f, g, and h be computable functions, let $C, x_1, x_2, ..., x_k$ be distinct program variables, and let φ represent formula*

$$f(x_1, ..., x_{i-1}, g(x_i), x_{i+1}, ..., x_k) = h(f(x_1, ..., x_k), x_1, ..., x_k)$$

which we call a differential identity for f. Then the following Hoare formula is satisfied:

$$\{\varphi \wedge C = f(x_1, ..., x_k)\}$$
$$C := h(C, x_1, ..., x_k)$$
$$x_i := g(x_i)$$
$$\{C = f(x_1, ..., x_k)\}$$

3.3 Notation

We use the following notation, taking advantage of the standard mathematical equivalence between binary relations and multi-valued functions. Let R and R' be binary relations and S be a set. Then:

$$\begin{aligned}
\mathsf{domain}(R) &= \{x | \exists y.\langle x, y\rangle \in R\} \\
\mathsf{range}(R) &= \{y | \exists x.\langle x, y\rangle \in R\} \\
R \triangleleft R' &= \{\langle x, z\rangle | \exists y.(\langle x, y\rangle \in R' \wedge \langle y, z\rangle \in R)\} \\
R^{-1} &= \{\langle y, x\rangle : \langle x, y\rangle \in R\} \\
R\{x\} &= \{y : \langle x, y\rangle \in R\} \\
R[S] &= \{y : \exists x \in S.\langle x, y\rangle \in R\} = \bigcup_{x \in S} R\{x\} \\
\bigcup S &= \bigcup_{x \in S} x
\end{aligned}$$

For example,

$$y \in R\{x\} \quad \text{iff} \quad x \in R^{-1}\{y\} \quad \text{iff} \quad \langle x, y\rangle \in R$$

We often use this notation for the transition relations $R_a = \left\{\langle p, p'\rangle : p \xrightarrow{a} p'\right\}$. Specifically, note that:

$$p' \in R_a\{p\} \quad \text{iff} \quad p \xrightarrow{a} p' \quad \text{iff} \quad p \in R_a^{-1}\{p'\}$$

Image set notation can be useful when data is naturally organized as a collection of sets, each of which is associated with a distinct value. It can also be useful when performance can be improved by accessing data via some fixed navigational path from the domain of a relation to its range. In both of these cases the image set $R\{x\}$ has a separate identity as a distinct data item. And this identity is especially apparent when individual image sets are updated by performing assignments of the following two kinds:

$R\{x\}$ `:=` $R\{x\} \cup \{y\}$ `--element addition`
$R\{x\}$ `:=` $R\{x\} - \{y\}$ `--element deletion`

However, when the image sets of R do not play any special role in the logical organization of data, and when a biased navigation serves no computational advantage, then we prefer to regard relations R in a more neutral way as a set of pairs. Under this interpretation, we can update R equivalently by performing the following assignments:

R := $R \cup \{\langle x, y\rangle\}$
R := $R - \{\langle x, y\rangle\}$

Furthermore, when R is a many-to-one relation, we use the conventional notation for function application $R(a)$ to denote the unique b such that $\langle a, b\rangle \in R$. $R(a)$ is undefined otherwise. In assignments,

$R(a)$:= b

means

$$R := \{\langle a', b'\rangle \in R | a' \neq a\} \cup \{\langle a, b\rangle\}$$

which preserves the single-valued nature of R.

Finally, it is useful to follow the SETL[43,42] convention of allowing boolean valued bounded existential quantifiers $\exists x \in S|K$ to have a side effect assignment. Whenever the quantifier evaluates to true, then variable x is assigned an arbitrary value that belongs to S and to the truth set of qualifier K. This convention allows us to to use such existential quantifiers as the condition of a while-loop whose body can refer to x in order to retrieve the truth set value newly chosen before each while-loop iteration.

4 Formal Specification

In order to decide whether one process is a ready simulation approximation of another, recall that whenever there exists a ready simulation relation $\leq$, there also exists a unique maximum one. Consequently, to decide whether two process are ready similar, we will compute the largest ready simulation relations in both directions, and then test whether the initial states of these processes are ready similar in both directions.

Let $P = \left\langle \Sigma_P, \dot{\rightarrow}_P, i_P \right\rangle$ and $Q = \left\langle \Sigma_P, \dot{\rightarrow}_Q, i_Q \right\rangle$ be two processes. The largest relation satisfying property (1) in our definition of ready simulation can be specified by,

$$E_1 = \max E \subseteq \Sigma_P \times \Sigma_Q \text{ such that}$$
$$\left(\forall \langle p, q\rangle \in E | \left\{a : \exists q' | q \xrightarrow{a} q' \in Q\right\} = \left\{a : \exists p' | p \xrightarrow{a} p' \in P\right\}\right)$$

which is a convenient initial approximation that overestimates the largest ready simulation relation.

We calculate E_1 by the the following method. Scan the edges in Q and P in linear time to obtain the sets of actions $\left\{a : \exists q' | q \xrightarrow{a} q' \in Q\right\}$ for each state q in Q, and $\left\{a : \exists p' | p \xrightarrow{a} p' \in P\right\}$ for each state p in P. Next, partition the states of $\Sigma_P \cup \Sigma_Q$ into equivalence classes of states that have the same set of actions. This partition ER_1 is a compact representation of E_1; *i.e.*,

$$E_1 = \bigcup \{(B \cap \Sigma_P) \times (B \cap \Sigma_Q) : B \in ER_1\}$$

The time to compute ER_1 is linear in the sum of the cardinalities of all the action sets by using multiset discrimination (see [36]).

Let R_a be the (nonempty) transition map for symbol a and processes P and Q; i.e.,

$$R_a = \left\{\langle r, r' \rangle | r \xrightarrow{a} r' \in P \vee r \xrightarrow{a} r' \in Q\right\}$$

and let A be the set of action symbols a that index these (nonempty) transition maps. Then the following is a specification for the largest ready simulation relation:

$$\begin{array}{l} \max E \subseteq E_1 \text{ such that} \\ \quad \left(\forall \langle p, q \rangle \in E, \forall a \in A, \forall \langle p, p' \rangle \in R_a, \exists \langle q, q' \rangle \in R_a | \langle p', q' \rangle \in E\right) \end{array} \tag{3}$$

Theorem 4.1 *Specification (3) is well-defined;* i.e., *there is always a unique maximum relation E satisfying the constraints.*

Proof: Use the equivalence $\left(\forall x \in S | \Phi(x)\right) \leftrightarrow \left(S = \{x \in S | \Phi(x)\}\right)$ to obtain the following specification equivalent to (3):

$$\begin{array}{l} \max E \subseteq E_1 \text{ such that} \\ \quad \left(E = \left\{\langle p, q \rangle \in E | (\forall a \in A, \forall p' \in R_a\{p\}, \exists q' \in R_a\{q\} | q' \in E\{p'\})\right\}\right) \end{array} \tag{4}$$

We simplify specification (4) by replacing predicate $\exists q' \in R_a\{q\} | q' \in E\{p'\}$ with $R_a\{q\} \cap E\{p'\} \neq \varnothing$, which is further reduced to $p' \in E^{-1}[R_a\{q\}]$. Predicate $\forall p' \in R_a\{p\} | p' \in E^{-1}[R_a\{q\}]$ is subsequently turned into the simpler but equivalent form $R_a\{p\} \subseteq E^{-1}[R_a\{q\}]$. The ready simulation specification that results from these simplifications is,

$$\max E \subseteq E_1 | \left(E = \left\{\langle p, q \rangle \in E | (\forall a \in A | R_a\{p\} \subseteq E^{-1}[R_a\{q\}])\right\}\right)$$

Since the function

$$F(E) = \left\{\langle p, q \rangle \in E | (\forall a \in A | R_a\{p\} \subseteq E^{-1}[R_a\{q\}])\right\} \tag{5}$$

```
E := E_1
while ∃ ⟨p_*, q_*⟩ ∈ (E − F(E)) loop
    % execution frequency O(n^2)
    E := E - {⟨p_*, q_*⟩}
end loop
```

Figure 2: First approximation to a ready simulation algorithm

is monotone in E, and since $(\wp(E_1), \cap, E_1)$ is a semilattice, we see that specification (3) is well-defined and equivalent to $\mathrm{gfp}(E = F(E))$ by Theorem 3.2. □

In order to support a coarse level of algorithmic analysis, we assume the following parameterization throughout this paper:

1. $m = \sum_{a \in A} |R_a|$, ; and
2. $n = \max(|\Sigma_P|, |\Sigma_Q|)$

This leads to an n^2 bound on the sizes of any binary relation defined on $\Sigma_P \bigcup \Sigma_Q$, including the input relation E_1 (and, of course, the output relation). However, it is important to observe that $|A|$ can vary from 0 to m, and that parameter m can vary from 0 to $2|A|n^2$.

5 Derivation of a Prototype Algorithm

Our derivation will proceed in two stages. First we derive a prototype algorithm that could be readily implemented in a high level programming language such as SETL[43,42]. Next, we use this prototype as the starting point for a derivation of a much more complex and efficient algorithm that could be implemented with somewhat greater effort in a conventional lower level language.

5.1 Dominated Convergence

By the proof of Theorem 4.1 and by Theorem 3.2, we could use function F defined in (5) to compute the largest ready simulation relation by $F^i(E_1)$ for some integer i. However, a high degree of redundancy makes this approach too inefficient. Fortunately, we can use Corollary 3.4 to compute $\mathrm{gfp}(E = F(E))$ more efficiently by performing the code shown in Figure 2. In the following two sections we will derive an efficient prototype algorithm from this initial program.

For clarity, we use special identifiers p_* and q_* for the bound variables of the existential quantifier at the top of the while-loop in Figure 2. Recall that

whenever the quantifier evaluates to true, then an arbitrarily chosen pair of values that belongs to $E - F(E)$ is assigned to p_* and q_* as a side effect. This pair $\langle p_*, q_* \rangle$ is subsequently deleted from E in the body of the loop. Our algorithmic analysis begins with the simple observation that E is updated at most $O(|E_1|) = O(n^2)$ times within the while-loop in Figure 2.

5.2 Finite Differencing

In this section we will recognize computational bottlenecks appearing in Figure 2, and remove them by maintaining program invariants. Our first goal is to maintain the invariant $W = E - F(E)$ appearing within the predicate of the while-loop. Note that W contains all those pairs of states in E that violate condition (1) in the definition of ready similar relations.

Before considering maintenance of W, it is useful to first simplify expression $E - F(E)$ into the equivalent expression

$$\{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - E^{-1}[R_a\{q\}] \neq \varnothing)\}$$

which can be rewritten more conveniently,

$$\{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - (E^{-1} \triangleleft R_a)\{q\} \neq \varnothing)\}$$

where

$$E^{-1} \triangleleft R_a = \{\langle qq, pp\rangle : \exists w.(\langle qq, w\rangle \in R_a \wedge pp \in E^{-1}\{w\})\}$$

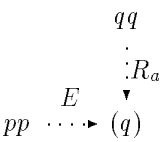
according to the set-theoretic definition of relation composition given earlier. In order to maintain the invariant

$$W = \{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - (E^{-1} \triangleleft R_a)\{q\} \neq \varnothing)\} \tag{6}$$

we will maintain invariants that keep values of subexpressions of (6) stored in program variables uniquely associated with these subexpressions. It is useful to consider these subexpressions from innermost to outermost order. The first such invariant is the relational composition

$$T1(a) = E^{-1} \triangleleft R_a$$

for each $a \in A$. For fixed $a \in A$ the preceding relational composition evaluates to the set of pairs $\langle qq, pp\rangle$



for which there exists a q such that $\langle qq, q\rangle$ belongs to R_a and $\langle pp, q\rangle$ belongs to E.

```
for aa ∈ A such that R_aa^{-1}{q_*} ≠ ∅ loop
    for qq_* ∈ R_aa^{-1}{q_*} loop
        % execution frequency O(mn)
        if |R_aa{qq_*} ∩ E{p_*}| = 1 then
            T1(aa){qq_*} := T1(aa){qq_*} - {p_*}
        end if
    end loop
end loop
```

Figure 3: Updating *T1*

Assume that for each $a \in A$ invariant *T1(a)* holds on entry to the while-loop of Figure 2. This invariant can be re-established immediately after it is spoiled as a result of the update

$$E := E - \{\langle p_*, q_* \rangle\}$$

inside the loop by executing the code in Figure 3 just before E is updated.

The innermost for-loop occurring in Figure 3 follows from the Finite Difference Schema of Lemma 3.6 using the differential identity

$$\begin{aligned} & (E - \{\langle p_*, q_* \rangle\})^{-1} \triangleleft R_a \\ = \quad & E^{-1} \triangleleft R_a - \left\{ \langle qq, p_* \rangle : qq \in R_a^{-1}\{q_*\} \middle| \left| R_a\{qq\} \cap E\{p_*\} \right| = 1 \right\} \end{aligned}$$

See Section A for a detailed proof that this works; such proofs are routine, and in the future we omit them. The intuitive idea is that removal of edge $\langle p_*, q_* \rangle$ from E will cause $E^{-1} \triangleleft R_a$ to lose edge $\langle qq_*, p_* \rangle$ iff the set $R_a\{qq_*\} \cap E\{p_*\}$ of nodes connecting qq_* and p_* only contains the single node q_*.

Consider an initial analysis of the run-time cost of executing Figure 3. Since each pair $\langle p_*, q_* \rangle$ of values can be selected only once at the top of the while-loop in Figure 2, then the number of times that $R_{aa}^{-1}\{q_*\}$ can be searched for each distinct pair of values q_* and aa is at most n in Figure 3. Hence, the total iteration count for this loop cannot exceed nm.

However, the outermost search through A and the repeated intersection calculation makes the code in Figure 3 too inefficient. These two problems are overcome by also maintaining the two invariants,

$$T2(aa) = \left\{ \langle \langle p, qq \rangle, q \rangle : \langle qq, q \rangle \in R_{aa}, p \in E^{-1}\{q\} \right\}$$

and

$$T3 = \{ \langle q, a \rangle : a \in A, q \in \mathsf{range}(R_a) \}$$

```
for aa ∈ T3{q*} loop
    % execution frequency O(mn)
    for qq* ∈ R_aa^{-1}{q*} loop
        % execution frequency O(mn)
        if |T2(aa){⟨p*, qq*⟩}| = 1 then
            T1(aa){qq*} := T1(aa){qq*} - {p*}
        end if
        T2(aa){⟨p*, qq*⟩} := T2(aa){⟨p*, qq*⟩} - {q*}
    end loop
end loop
```

Figure 4: Maintaining invariants for *T1* and *T2*

In particular, invariant $T3$ improves the search through A, and invariant $T2(aa)$ implies invariant

$$T2(aa)\{\langle p_*, qq_*\rangle\} = R_{aa}\{qq_*\} \cap E\{p_*\}$$

so that the the intersection $R_{aa}\{qq_*\} \cap E\{p_*\}$ need not be calculated repeatedly. That is, just prior to the change in E, we can maintain both invariants *T1(aa)* and *T2(aa)* together by executing the code in Figure 4 instead of the code in Figure 3. The code to update *T2(aa)* follows from the Finite Difference Schema Lemma 3.6 using the differential identity,

$$\begin{aligned} & \left\{\langle\langle p, qq\rangle, q\rangle : \langle qq, q\rangle \in R_{aa}, p \in \left(E - \{\langle p_*, q_*\rangle\}\right)^{-1}\{q\}\right\} \\ = \quad & \left\{\langle\langle p, qq\rangle, q\rangle : \langle qq, q\rangle \in R_{aa}, p \in E^{-1}\{q\}\right\} - \left\{\langle\langle p_*, qq\rangle, q_*\rangle : qq \in R_{aa}^{-1}\{q_*\}\right\} \end{aligned}$$

Now we can consider how to maintain the top-level invariant,

$$W = \{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\}$$

relative to changes in both E and *T1*. Just before pair $\langle p_*, q_*\rangle$ (which was originally selected arbitrarily from W) is removed from E, we can update W by executing

$$W := W - \{\langle p_*, q_*\rangle\}$$

Just prior to the change

$T1(aa)\{qq_*\} := T1(aa)\{qq_*\} - \{p_*\}$

we can update W by executing the code shown in Figure 5, which can be made more efficient by maintaining the additional invariant,

$$WC = E - W$$

```
for pp* ∈ R_aa^{-1}{p*} loop
    % execution frequency O(mn)
    if ⟨pp*, qq*⟩ ∈ (E − W) then
        W := W ∪ {⟨pp*, qq*⟩}
    end if
end loop
```

Figure 5: Updating W

Since the predicate $|T2(aa)\{\langle p_*, qq_*\rangle\}| = 1$, appearing in Figure 4, will be true at most once for each distinct triple aa, p_* and qq_*, then for each distinct value of aa, the predicate is satisfied no more than n times for each value of p. Hence, for each distinct value of aa, we know that $R_{aa}^{-1}\{p_*\}$ can be searched for fixed p_* at most n times. Consequently, the execution frequency of the body of the for-loop in Figure 5 is at most $O(nm)$.

The update for W relative to the modifications to $T1(aa)\{qq_*\}$ follows from the Finite Difference Schema and the differential identity,

$$\left\{\langle p,q\rangle \in E \middle| \left(\exists a \in A | (R_a\{p\} - \left(\begin{array}{l}\text{if } q = qq_* \wedge aa = a \\ \quad \text{then } T1(aa)\{qq_*\} - \{p_*\} \\ \quad \text{else } T1(a)\{q\}\end{array}\right)) \neq \varnothing\right)\right\}$$

$$= \{\langle p,q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\} \cup$$
$$\left\{\langle p, qq_*\rangle : p \in R_{aa}^{-1}\{p_*\} | \langle p, qq_*\rangle \in E \wedge (\nexists a \in A | R_a\{p\} - T1(a)\{qq_*\} \neq \varnothing)\right\}$$

which is obtained as follows. First we calculate,

$$\left(R_a\{p\} - \left(\begin{array}{l}\text{if } q = qq_* \wedge aa = a \\ \quad \text{then } T1(aa)\{qq_*\} - \{p_*\} \\ \quad \text{else } T1(a)\{q\}\end{array}\right)\right) \neq \varnothing$$

$$= \left(\begin{array}{l}\text{if } q = qq_* \wedge aa = a \\ \quad \text{then } \big(R_a\{p\} - (T1(a)\{q\} - \{p_*\}) \neq \varnothing\big) \\ \quad \text{else } (R_a\{p\} - T1(a)\{q\} \neq \varnothing)\end{array}\right)$$

$$= \left(\begin{array}{l}\text{if } q = qq_* \wedge aa = a \\ \quad \text{then } \big((R_a\{p\} - T1(a)\{q\}) \cup (R_a\{p\} \cap \{p_*\}) \neq \varnothing\big) \\ \quad \text{else } (R_a\{p\} - T1(a)\{q\} \neq \varnothing)\end{array}\right)$$

$$= (R_a\{p\} - T1(a)\{q\} \neq \varnothing) \vee (a = aa \wedge q = qq_* \wedge p \in R_a^{-1}\{p_*\})$$

The rest of the derivation proceeds as follows.

$$\left\{\langle p,q\rangle \in E \middle| \left(\exists a \in A | (R_a\{p\} - \left(\begin{array}{l}\text{if } q = qq_* \wedge aa = a \\ \quad \text{then } T1(aa)\{qq_*\} - \{p_*\} \\ \quad \text{else } T1(a)\{q\}\end{array}\right)) \neq \varnothing\right)\right\}$$

$$= \left\{\langle p,q\rangle \in E | (\exists a \in A | (R_a\{p\} - T1(a)\{q\} \neq \varnothing \vee (a = aa \wedge q = qq_* \wedge p \in R_a^{-1}\{p_*\})))\right\}$$

$$= \left\{\langle p,q\rangle \in E | (\exists a \in A | (R_a\{p\} - T1(a)\{q\} \neq \varnothing) \vee (\exists a \in A | (a = aa \wedge q = qq_* \wedge p \in R_a^{-1}\{p_*\})))\right\}$$

$$= \left\{\langle p,q\rangle \in E | \left(\exists a \in A | \begin{array}{l}(R_a\{p\} - T1(a)\{q\} \neq \varnothing) \vee \\ (q = qq_* \wedge p \in R_{aa}^{-1}\{p_*\} \wedge \nexists a \in A | R_a\{p\} - T1(a)\{qq_*\} \neq \varnothing)\end{array}\right)\right\}$$

$$= \{\langle p,q\rangle \in E | (\exists a \in A | (R_a\{p\} - T1(a)\{q\} \neq \varnothing)\} \cup$$
$$\left\{\langle p,q\rangle \in E | q = qq_* \wedge p \in R_{aa}^{-1}\{p_*\} \wedge (\nexists a \in A | R_a\{p\} - T1(a)\{qq_*\} \neq \varnothing)\right\}$$

$$= \{\langle p,q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\} \cup$$
$$\left\{\langle p,qq_*\rangle : p \in R_{aa}^{-1}\{p_*\} | \langle p,qq_*\rangle \in E \wedge (\nexists a \in A | R_a\{p\} - T1(a)\{qq_*\} \neq \varnothing)\right\}$$

Observe that invariant WC and W can be maintained collectively by executing the following code just before the modification to $T1$,

```
for pp* ∈ R_aa^{-1}{p*} loop
    % execution frequency O(mn)
    if ⟨pp*, qq*⟩ ∈ WC then
        % execution frequency O(n^2)
        WC := WC - {⟨pp*, qq*⟩}
        W := W ∪ {⟨pp*, qq*⟩}
    end if
end loop
```

WC does not have to be updated just prior to the element deletion from E, since the deleted element belonged to W and not WC.

Putting the preceding code fragments together, and noting that $T1$ is never used, that a set of reference counts

$$T2count(aa)(p_*, qq_*) = \left|T2(aa)\{\langle p_*, qq_*\rangle\}\right|$$

can replace the stored sets $T2(aa)\{\langle p_*, qq_*\rangle\}$, and that $WC = E$ on exit from the while-loop, we obtain the prototype code in Figure 6.

To make an initial performance analysis of the code in Figure 6, we can assume that all element additions, element deletions, membership tests, and arbitrary selection operations (*e.g.*, the exists predicate) take unit time, and searching through a set X takes time linear in the size of X. Certainly, universal hashing [14] will achieve these times in a probabilistic sense. Later, we will transform these set operations into lower level code that achieves these time bounds in the worst case. Under these assumptions, it is apparent that the time

```
while ∃ ⟨p*, q*⟩ ∈ W loop
    for aa ∈ T3{q*} loop
        for qq* ∈ R_aa^-1{q*} loop
            if T2count(aa)(p*, qq*) = 1 then
                for pp* ∈ R_aa^-1{p*} loop
                    if ⟨pp*, qq*⟩ ∈ WC then
                        WC := WC - {⟨pp*, qq*⟩}
                        W := W ∪ {⟨pp*, qq*⟩}
                    end if
                end loop
            end if
            T2count(aa)(p*, qq*) := T2count(aa)(p*, qq*) − 1
        end loop
    end loop
    W := W - {⟨p*, q*⟩}
end loop
% WC is the desired ready simulation relation.
```

Figure 6: Prototype code

bounds depend entirely on the total number of while-loop iterations and the total number of for-loop iterations through $R_{aa}^{-1}\{q_*\}$ and through $R_{aa}^{-1}\{p_*\}$. By previous analysis, the worst case time for the code in Figure 6 is $O(mn + n^2)$.

5.3 Establishing Invariants

To complete the derivation of a prototype algorithm, we need to discuss how to establish the invariants for *T1*, *T2count*, *T3*, W, and WC on entry to the while-loop, where E_1 is assigned to E. The method to establish these invariants involves loop fusion similar to the approach found in [38].

We can establish $T3 = \{\langle q, a\rangle : a \in A, q \in \mathsf{range}(R_a)\}$ by executing,

```
T3 := ∅
for a ∈ A loop
    % execution frequency O(|A|) = O(m)
    for q* ∈ domain(R_a^{-1}) loop
        % execution frequency O(m)
        T3{q*} := T3{q*} ∪ {a}
    end loop
end loop
```

Subsequently, we can establish $T1(a) = E^{-1} \triangleleft R_a$ and *T2count* using a search through *T3* according to the code in Figure 7. This code is obtained by reducing the problem of establishing invariants *T1* and *T2count* to the problem of maintaining them, first with respect to an assignment $E := \varnothing$, and then with respect to successively adding each pair $\langle p_*, q_* \rangle$ that belongs to E_1 to E. The maintenance code relative to adding pairs to E is the dual form of maintenance code relative to the deletions of such pairs, and is justified in the same way as the code found in Figure 4. Arguments to support the $O(mn)$ bound on the frequencies of execution shown in Figure 7 are dual to those previously given for the code found in Figure 4.

```
T2count := ∅
T1 := ∅
for ⟨p*, q*⟩ ∈ E1 loop
    % execution frequency O(n^2)
    for a ∈ T3{q*} loop
        % execution frequency O(mn)
        for qq* ∈ R_a^{-1}{q*} loop
            % execution frequency O(mn)
            % assume T2count(a)(p*, qq*) = 0 if undefined
            if T2count(a)(p*, qq*) = 0 then
                T1(a){qq*} := T1(a){qq*} ∪ {p*}
            end if
            T2count(a)(p*, qq*) := T2count(a)(p*, qq*) + 1
        end loop
    end loop
end loop
```

Figure 7: Code to establish *T1* and *T2count*

In order to establish

$$W = \{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\},$$

we will make use of the three new invariants,

$$wcount2(a)(p, q) = |R_a\{p\} - T1(a)\{q\}|,$$

for all $a \in A$ and $\langle p, q\rangle \in E$ such that $R_a\{p\} \neq \varnothing$,

$$T4 = \{\langle p, a\rangle : a \in A, p \in \mathsf{domain}(R_a)\},$$

and

$$wcount(p, q) = \sum_{a \in T4\{p\}} wcount2(a)(p, q).$$

Based on these invariants, we see that

$$\begin{aligned} W &= \{\langle p, q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\} \\ &= \{\langle p, q\rangle \in E | (\exists a \in A | wcount2(a)(p, q) > 0)\} \\ &= \{\langle p, q\rangle \in E | wcount(p, q) > 0\} \end{aligned}$$

We also know that for all $a \in A$, $[R_a\{p\} \neq \varnothing$ iff $a \in T4\{p\}]$.

```
T3 := ∅
T4 := ∅
for a ∈ A loop
    for q* ∈ range(R_a) loop
        % execution frequency O(m)
        T3{q*} := T3{q*} ∪ {a}
    end loop
    for q* ∈ domain(R_a) loop
        % execution frequency O(m)
        T4{q*} := T4{q*} ∪ {a}
    end loop
end loop
```

Figure 8: Code to establish *T3* and *T4*

In order to establish these invariants, we first recognize that *T4* can be established efficiently together with *T3*; the code is given in Figure 8. The

computation to establish the remaining invariants is broken into two parts for convenience. If we assume that $T1(a) = \varnothing$ for all $a \in A$, then

$$\begin{aligned} W &= \{\langle p,q\rangle \in E | (\exists a \in A | R_a\{p\} - T1(a)\{q\} \neq \varnothing)\} \\ &= \{\langle p,q\rangle \in E | (\exists a \in A | R_a\{p\} \neq \varnothing)\} \\ &= \{\langle p,q\rangle \in E | T4\{p\} \neq \varnothing\} \end{aligned}$$

and

$$wcount2(a)(p,q) = |R_a\{p\}|$$

for all $a \in A$ and $\langle p,q\rangle \in E$ such that $R_a\{p\} \neq \varnothing$.

Using the preceding identities we can establish initial approximations of invariants W, WC, *wcount2*, and *wcount* by the code in Figure 9. This code arises by first maintaining these invariants (under the assumption that $T1$ is empty) with respect to assignment $E := \varnothing$, and then maintaining them with respect to successively adding each pair $\langle p_*, q_*\rangle$ in E_1 to E.

Next, we adjust *wcount*, W, and WC to take the actual value of *T1* into account. This code, which is given in Figure 10, arises from maintaining the three invariants relative to imaginary operations of the form $T1(a)\{qq_*\} := T1(a)\{qq_*\} \cup \{p_*\}$ that augment $T1$ (assumed to be empty initially) repeatedly until it reaches its correct value. The execution frequency of the inner for-loop within Figure 10 is justified in a way dual to the argument supporting the execution frequency appearing in Figure 5. That is, although we treat $T1$ as being augmented here but diminished in the earlier case, the worst case number of times it is modified one way or the other must be of the same asymptotic order.

Observe finally, that WC, E_1, and E can be equivalenced into the single variable WC, that $T1(a)^{-1} = \mathsf{domain}(T2count(a))$ (so that *T1(a)* can be eliminated), that W and domain *wcount* can be equivalenced, and that the first loop above (which is used to partially establish W, WC, *wcount2(a)*, and *wcount*) can be jammed into the same loop that establishes *T1(a)* and *T2count*. Consequently, we can establish all the invariants needed within the main while-loop shown in Figure 6 by executing the code in Figure 11.

Under the same assumptions as was used to analyze the while-loop, we see that the preceding code takes $O(mn + n^2)$ time.

6 Derivation of a RAM Implementation

The critical task of an efficient implementation is to provide data structures supporting unit-time associative access to $\mathsf{domain}(wcount2(a))$, $\mathsf{domain}(T2count(a))$, $\mathsf{domain}\left(R_a{}^{-1}\right)$, WC, $\mathsf{domain}(wcount)$, $\mathsf{domain}(T4)$, and $\mathsf{domain}(T3)$. We also need to support searching through a set or through the domain of a map in

```
wcount := ∅; wcount2 := ∅
for a ∈ A loop
    wcount2(a) := ∅
end loop
W := ∅
WC := E_1
for ⟨p_*, q_*⟩ ∈ E_1 loop
    if T4{p_*} ≠ ∅ then
        WC := WC - {⟨p_*, q_*⟩}
        W := W ∪ {⟨p_*, q_*⟩}
        for a ∈ T4{p_*} loop
            % execution frequency O(nm)
            % assume wcount(p_*, q_*) = 0 if undefined
            wcount(p_*, q_*) := wcount(p_*, q_*) + |R_a{p_*}|
            wcount2(a)(p_*, q_*) := |R_a{p_*}|
        end loop
    end if
end loop
```

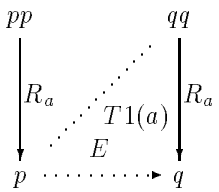


Figure 9: Establishing W, WC, *wcount*, and *wcount2* ignoring *T1*

```
for a ∈ A loop
    for ⟨qq*, p*⟩ ∈ T1(a) loop
        for pp* ∈ R_a^{-1}{p*} loop
            % execution frequency O(nm)
            if wcount2(a)(pp*, qq*) > 0 then
                if wcount(pp*, qq*) = 1 then
                    W := W - {⟨pp*, qq*⟩}
                    WC:= WC ∪ {⟨pp*, qq*⟩}
                end if
                %make wcount(pp*, qq*) undefined if it drops to 0
                wcount(pp*, qq*) := wcount(pp*, qq*) - 1
                wcount2(a)(pp*, qq*) := wcount2(a)(pp*, qq*) - 1
            end if
        end loop
    end loop
end loop
```

Figure 10: Establishing *W*, *WC*, *wcount*, and *wcount2* considering *T1*

time proportional to its cardinality, unit-time element addition to a set or to the domain of a map, and unit-time element deletion from a set or from the domain of a map. The method applied here is based on the real-time simulation of a set machine on a RAM [35]. Although the method is theoretically sound for any transition relation, we will tailor it to obtain an implementation that performs especially well when the transition relation is sparse, which is by far the most common case.

Unit-time associative access is achieved by array access using integer encodings for the symbols in A and the states in Σ_P and Σ_Q. That is, we assume unique integer identifiers from 1 to $|\Sigma_P|$ for the states in Σ_P, and unique integer identifiers from 1 to $|\Sigma_Q|$ for the states in Σ_Q. We also assume integer identifiers from 1 to $|A|$ for the symbols in A. For each symbol a belonging to alphabet A, we store three arrays – one dimensional arrays P_a and Q_a of size $|\Sigma_P|$ and $|\Sigma_Q|$ respectively, and two-dimensional array PQ_a of size $|\Sigma_P| \times |\Sigma_Q|$ (see Figure 12).

By using a linked list to store the elements of a set or domain of a map, linear time search is achieved. Unit-time element deletion is achieved by making the list doubly linked. Unit-time element addition to a set is easily implemented as list insertion, because all of our transformations guarantee that the set will never contain the element to be added (so a run-time search to test membership can be avoided).

For each Σ_P state i, the component $P_a(i)$ is a record with two fields – the first storing the count $|R_a\{i\}|$ and the second storing a pointer to ${R_a}^{-1}\{i\}$, where

```
T3 := ∅; T4 := ∅; wcount := ∅; wcount2 := ∅; T2count := ∅
for a ∈ A loop
    wcount2(a) := ∅
    T2count(a) := ∅
    for q* ∈ range(R_a) do T3{q*} := T3{q*} ∪ {a}
    for q* ∈ domain(R_a) do T4{q*} := T4{q*} ∪ {a}
end loop
WC := E_1% pointer assignment can be used for efficiency
for ⟨p*, q*⟩ ∈ WC loop
    for a ∈ T3{q*} loop
        % assume T2count(a)(p*, qq*) = 0 if undefined
        for qq* ∈ R_a^{-1}{q*} do T2count(a)(p*, qq*) := T2count(a)(p*, qq*) + 1
    end loop
    if T4{p*} ≠ ∅ then
        WC := WC - {⟨p*, q*⟩}
        for a ∈ T4{p*} loop
            %assume wcount(p*, q*) = 0 if undefined
            wcount(p*, q*) := wcount(p*, q*) + |(R_a{p*})|
            wcount2(a)(p*, q*) := |(R_a{p*})|
        end loop
    end if
end loop
for a ∈ A loop
    for ⟨p*, qq*⟩ ∈ domain T2count(a) loop
        for pp* ∈ R_a^{-1}{p*} loop
            if wcount2(a)(pp*, qq*) > 0 then
                if wcount(pp*, qq*) = 1 do WC:= WC ∪ {⟨pp*, qq*⟩}
                %make wcount(pp*,qq*) undefined if it drops to 0
                wcount(pp*, qq*) := wcount(pp*, qq*) - 1
                wcount2(a)(pp*, qq*) := wcount2(a)(pp*, qq*) - 1
            end if
        end loop
    end loop
end loop
W := domain(wcount) % pointer assignment can be used for efficiency
```

Figure 11: Full Initialization

set $R_a{}^{-1}\{i\}$ is a one-way list of integers designating Σ_P-states. For each Σ_Q state i, component $Q_a(i)$ is a record with only one field – a pointer to $R_a{}^{-1}\{i\}$, where set $R_a{}^{-1}\{i\}$ is a one-way list of integers designating Σ_Q-states. Hence, these pre-images can be searched in time proportional to their cardinalities.

Array PQ_a is used to store *wcount2(a)* and *T2count(a)*. For each Σ_P-state i and Σ_Q-state j, $PQ_a(i,j)$ is a record with five fields. One of these fields stores $T2count(a)(i,j)$, and another stores *wcount2(a)(i,j)*. Another field stores a pointer that links up all those records in PQ_a with nonempty *T2count(a)* fields. The two other fields store states i and j (which are cursors to be used for array access). The list linking records with nonempty *T2count(a)* fields supports the linear time search through domain($T2count(a)$) in the last loop of initialization (cf. Figure 11). Within this loop the two cursor fields (which represent states p_* and qq_*) are used to access other data efficiently.

A one dimensional master symbol array M of size $|A|$ is used to access the three arrays P_a, Q_a, and PQ_a for any symbol a belonging to A. For each symbol a, component *M(a)* is a record with three fields, a pointer to P_a, a pointer to Q_a, and a pointer to PQ_a.

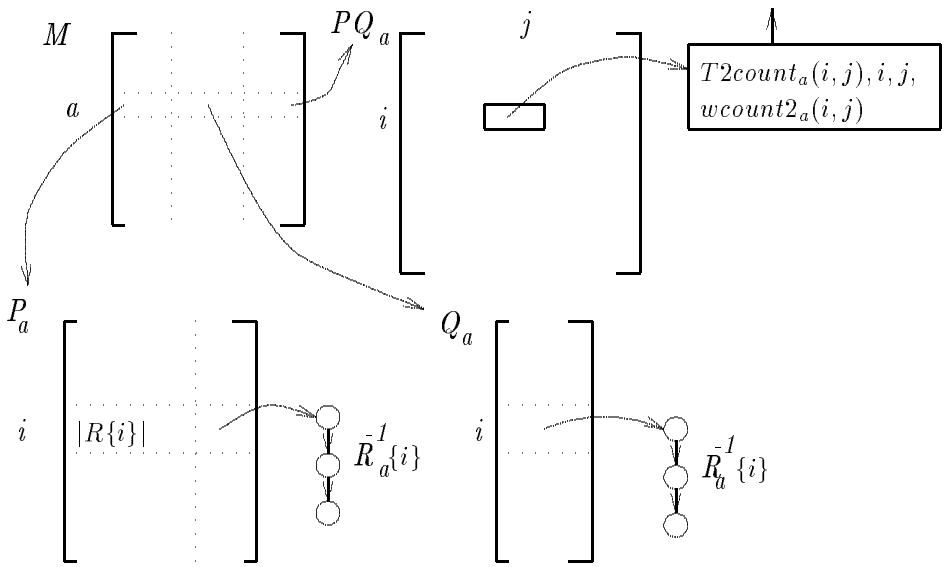


Figure 12: Data Structures for $T2count_a$, $wcount2_a$, and $R_a{}^{-1}$ for each $a \in A$.

There are three more arrays – one dimensional arrays P' and Q' of size $|\Sigma_P|$ and $|\Sigma_Q|$ respectively, and two-dimensional array PQ' of size $|\Sigma_P| \times |\Sigma_Q|$. For each Σ_P state i, component $P'(i)$ stores one field – a pointer to $T4\{i\}$, where set $T4\{i\}$ is a one-way list of integers designating symbols in A and used to

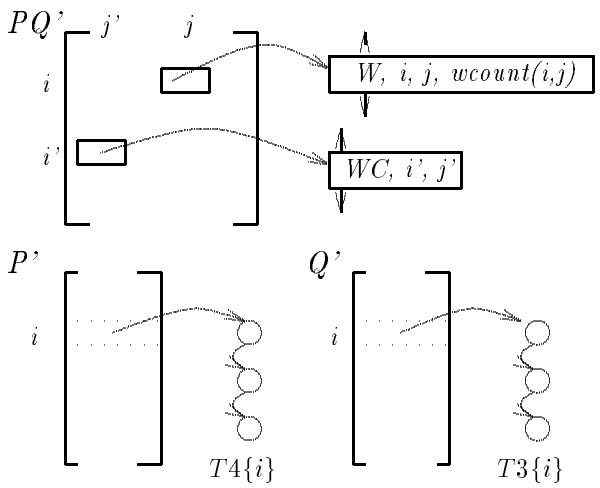


Figure 13: Data Structures For W, *wcount*, WC, *T3*, and *T4*.

access array M. Similarly, for each Σ_Q state i, component $Q'(i)$ stores one field – a pointer to $T3\{i\}$, where set $T3\{i\}$ is a one-way list of integers designating symbols in A and used to access array M.

For each Σ_P-state i and Σ_Q-state j, component $PQ'(i,j)$ stores one of two kinds of records. One kind stores pairs of states that belong to W; the other kind stores pairs of states that belong to WC. Both kinds of records can be stored conveniently in the same matrix, because W and WC are disjoint. For each pair of states $\langle i', j' \rangle$ that belongs to WC there is a WC record that contains four fields. Two of these fields store states i' and j' (which are cursors to be used for array access). The other two fields contain forward and backward pointers that connect to a doubly linked list of all PQ'-array components storing WC records. For each pair of states $\langle i, j \rangle$ that belongs to W there is a W record that contains five fields. One field stores count *wcount(i,j)*. Two other fields are used to store states i and j (which are cursors to be used for array access). The other two fields contain forward and backward pointers that connect to a doubly linked list of all PQ'-array components storing W records. We will also need pointers to one end of the doubly linked lists storing W records and to both ends of the list storing WC records (see Figure 13).

It is worthwhile discussing how the preceding implementations for W and WC can be used to perform the set theoretic operations of our final prototype ready simulation program efficiently. The doubly linked list implementation of WC supports linear time searches using the for-loop within the initialization (cf. Figure 11) and for printing the final output (not shown). It also supports unit-

time element deletions appearing in the initialization code and in the center of the main while-loop (cf. Figure 6). Array access is needed to obtain a unit-time membership test inside the while-loop.

Implementation of W together with *wcount* (recall that W and domain *wcount* are aliased) is slightly more complicated. The implementation of W as a doubly linked list allows arbitrary pairs to be retrieved and subsequently deleted in unit time from one end of the list via a pointer to that end (cf. the main while-loop in Figure 6). Unit-time element addition to W within the main while-loop is performed by list insertion to the other end of the list via a pointer to that end. The double linking supports an implicit unit-time deletion from domain *wcount* whenever the operation $wcount(pp_*, qq_*) := wcount(pp_*, qq_*) - 1$, appearing in the last for-loop of the initialization (cf. Figure 11), resets the count $wcount(pp_*, qq_*)$ to zero. Array access is needed to perform the map application and dynamic map assignment (which both entail associative access to domain *wcount*) to *wcount* in unit time within the final for-loop of the initialization.

The total space for all of the preceding data structures is $O(|A| \times (|\Sigma_P| + |\Sigma_Q| + |\Sigma_P| \times |\Sigma_Q|))$. In case these three arrays are sparsely occupied, then each array can be initialized in unit time using the solution to [1, exercise 18, p.100]. Although the main implementation details have been discussed, several issues such as how to read external input for E_1 and maps ${R_a}^{-1}$ for each $a \in A$ have been omitted, and are left to the reader.

7 Conclusion

This paper has demonstrated the viability of a transformational methodology by systematically using it to derive a new algorithm and simultaneously to construct an efficient implementation starting from a mathematical specification of the ready simulation problem. Aspects of algorithmic analysis were formally integrated into the derivation, and helped to guide the transformational steps.

We implemented the prototype algorithm in Common Lisp and in SETL2, and both implementations performed acceptably even using general-purpose Common Lisp and SETL2 primitives rather than the more efficient RAM data structures. Details on how to obtain and run the SETL2 implementation are found in Appendix B.

Although we have not implemented the final RAM algorithm, it should also be straightforward to do so in any conventional language. Although the final implementation was derived by transformation, we believe that its performance would be competitive with any tightly coded hand crafted program.

The 'natural' style of the ready simulation derivation (in the way it captures algorithm design principles within a succinct syntactic mechanism) suggests that this transformational methodology could be used more generally as part of a manual programming methodology. The goal directed nature of the derivation and the simplicity of the derivational steps also suggest the possibility of a

semi-automatic approach in which major portions of the derivation could be fully mechanized.

8 Acknowledgements

We thank Paul Taylor for the use of the Diagrams package.

A Sample Correctness Proof

In this section, we present a detailed proof that the code in Figure 3 is correct. We state our conditions in Hoare logic, though our reasoning is largely semantic.

Let

$$\Phi = \forall a \in A | T1(a) = E^{-1} \triangleleft R_a$$

be the condition that *T1* has the correct value. In Hoare logic, we wish to show that Φ is an invariant:

$$\{\Phi\}\text{Figure 3}; E := E - \{\langle p_*, q_* \rangle\}\{\Phi\}$$

A.1 Eliminating Loops

In Figure 3, we repeatedly use `for` loops over unordered sets. In general, the Hoare logic of such loops is nontrivial, as the invariant must take into account the order in which the set elements actually appear. However, our loops are essentially parallel: they update separate components of data structures. We therefore transform our looping, nondeterministically-executed program into an equivalent non-looping, deterministic program working with larger data structures. This makes the Hoare computation trivial, but does require some lemmas allowing us to transform programs.

We say that two program fragments S and S' are *Hoare equivalent* if for every two formulas φ and φ', $\{\varphi\}S\{\varphi'\}$ iff $\{\varphi\}S'\{\varphi'\}$.

Lemma A.1 *Let T be a multi-valued map variable. If Q is a set-valued expression, if $P(u,v)$ is a predicate, and if variables y, x, and T do not occur in either Q or P, then the program fragment*

```
for x in Q loop
    if P(x,y) then
        T{x} := T{x} - y
    end if
end loop
```

is Hoare equivalent *to the program*

T := T - $\{\langle x, y \rangle | x \in Q(x) \wedge P(x,y)\}$

(Note that y is free in both program fragments.)

Proof: Straightforward; both programs have the same input/output relation semantics. The condition that T, x, and y do not appear in expressions Q and P guarantees that the order of performing the loop does not matter. □

Lemma A.2 *Let T be a single-valued map variable. If Q is a set-valued expression whose value is contained in the domain of T, if $e(u)$ is an expression, and if variables T and x do not appear in Q or e, then the program*

```
for x in Q loop
    T(x) := T(x) - e(x)
end loop
```

is Hoare equivalent to the program

$$T := \{\langle x, T(x)\rangle | x \in \mathsf{domain}(T) \wedge x \notin Q\} \cup \{\langle x, T(x) - e(x)\rangle | x \in \mathsf{domain}(T) \wedge x \in Q\}$$

These lemmas can be used to rewrite the code of Figure 3 as a single assignment. This is achieved by first invoking Lemma A.1 to replace the Figure 3 code just below,

```
for a in A such that R_a^{-1}{q_*} ≠ ∅ loop
    for qq_* in R_a^{-1}{q_*} loop
        if |R_a{qq_*} ∩ E{p_*}| = 1 then
            T1(a){qq_*} := T1(a){qq_*} - {p_*}
        end if
    end loop
end loop
```

by the simpler Hoare equivalent code,

```
for a in A such that R_a^{-1}{q_*} ≠ ∅ loop
    T1(a) := T1(a) - {⟨qq, p_*⟩ | qq ∈ R_a^{-1}{q_*} ∧ |R_a{qq} ∩ E{p_*}| = 1}
end loop
```

Since $\left\{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left(\left|R_a\{qq\} \cap E\{p_*\}\right| = 1\right)\right\}$ is empty whenever $R_a^{-1}\{q_*\}$ is, then the code just above can be simplified to,

```
for a in A loop
    T1(a) := T1(a) - {⟨qq, p_*⟩ | qq ∈ R_a^{-1}{q_*} ∧ |R_a{qq} ∩ E{p_*}| = 1}
end loop
```

which, by Lemma A.2, is Hoare equivalent to,

$$T1 := \{\langle a, T1(a)\rangle | a \in \mathsf{domain}(T1) \wedge a \notin A\} \cup$$
$$\left\{\left\langle a, T1(a) - \{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left|R_a\{qq\} \cap E\{p_*\}\right| = 1\}\right\rangle | a \in \mathsf{domain}(T1) \wedge a \in A\right\}$$

Finally, since $A = \mathsf{domain}(T1)$, we can simplify the code just above to the single assignment,

$$T1 \;:=\; \left\{\left\langle a,\, T1(a) - \left\{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left|R_a\{qq\} \cap E\{p_*\}\right| = 1\right\}\right\rangle | a \in A\right\}$$

Call this command S.

A.2 Verifying S

Theorem A.3 *The Hoare formula* $\{\Phi\}S; E := E - \{\langle p_*, q_*\rangle\}\{\Phi\}$ *is satisfied.*

Proof: The theorem is an instance of finite differencing Lemma 3.6, where:

$$\begin{aligned}
k &= 1 \\
C &= T1 \\
x_1 &= E \\
f(E) &= \left\{\left\langle a, E^{-1} \triangleleft R_a\right\rangle | a \in A\right\} \\
g(E) &= E - \{\langle p_*, q_*\rangle\} \\
h(T1, E) &= \left\{\left\langle a, T1(a) - \left\{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left(\left|R_a\{qq\} \cap E\{p_*\}\right| = 1\right)\right\}\right\rangle | a \in A\right\} \\
\varphi &= f(g(E)) = h(f(E), E)
\end{aligned}$$

By Lemma 3.6, it suffices to show that $f(g(E)) = h(f(E), E)$; that is,

$$\begin{aligned}
& \left\{\left\langle a, \left(E - \{\langle p_*, q_*\rangle\}\right)^{-1} \triangleleft R_a\right\rangle | a \in A\right\} \\
= \;& \left\{\left\langle a, E^{-1} \triangleleft R_a - \left\{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left|R_a\{qq\} \cap E\{p_*\}\right| = 1\right\}\right\rangle | a \in A\right\}
\end{aligned}$$

which is true iff these two functions are pointwise equal; that is, for all a in A,

$$\left(E - \{\langle p_*, q_*\rangle\}\right)^{-1} \triangleleft R_a = E^{-1} \triangleleft R_a - \left\{\langle qq, p_*\rangle | qq \in R_a^{-1}\{q_*\} \wedge \left|R_a\{qq\} \cap E\{p_*\}\right| = 1\right\}$$

We show this by calculation; i.e.,

$$\begin{aligned}
& \left(E - \{\langle p_*, q_*\rangle\}\right)^{-1} \triangleleft R_a \\
= \;& \left\{\langle q, pp\rangle | (\exists qq | (\langle q, qq\rangle \in R_a \wedge \langle qq, pp\rangle \in \left(E - \{\langle p_*, q_*\rangle\}\right)^{-1}))\right\} \\
= \;& \left\{\langle q, pp\rangle | R_a\{q\} \cap \left(E - \{\langle p_*, q_*\rangle\}\right)\{pp\} \neq \varnothing\right\} \\
= \;& \left\{\langle q, pp\rangle | (R_a\{q\} \cap E\{pp\}) - (R_a\{q\} \cap (\{\langle p_*, q_*\rangle\}\{pp\})) \neq \varnothing\right\} \\
= \;& \left\{\langle q, pp\rangle | (R_a\{q\} \cap E\{pp\}) \neq \varnothing \wedge (pp = p_* \wedge q_* \in R_a\{q\} \Rightarrow \left|R_a\{q\} \cap E\{pp\}\right| \neq 1)\right\} \\
= \;& \left\{\langle q, pp\rangle | R_a\{q\} \cap E\{pp\}) \neq \varnothing \wedge \neg(pp = p_* \wedge q_* \in R_a\{q\} \wedge \left|R_a\{q\} \cap E\{pp\}\right| = 1)\right\} \\
= \;& (E^{-1} \triangleleft R_a) - \left\{\langle q, pp\rangle | pp = p_* \wedge q_* \in R_a\{q\} \wedge \left|R_a\{q\} \cap E\{pp\}\right| = 1\right\} \\
= \;& (E^{-1} \triangleleft R_a) - \left\{\langle q, p_*\rangle | q \in R_a^{-1}\{q_*\} \wedge \left|R_a\{q\} \cap E\{p_*\}\right| = 1\right\}
\end{aligned}$$

□

B Sample Implementation

A SETL2 implementation of the prototype algorithm is available on the World-Wide Web. The program itself is available in the software section of the home page,

`http://cs.nyu.edu/cs/faculty/paige/`

A free implementation of SETL2 is available by anonymous FTP at

`gopher://cs.nyu.edu/11/Computer-Science-Department-Anonymous-FTP/languages/setl2`

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[3] D. Austry and G. Boudol. Algèbre de processus et synchronisation. *Theoretical Computer Sci.*, 30(1):91–131, 1984.

[4] C. Beeri and P. Bernstein. Computational problems related to the design of normal form relation schemes. *ACM Trans. Database Syst.*, 4(1):30–59, 1979.

[5] J. A. Bergstra and J. W. Klop. Process algebra for synchronous communication. *Information and Computation*, 60(1–3):109–137, 1984.

[6] B. Bloom. *Ready Simulation, Bisimulation, and the Semantics of CCS-Like Languages*. PhD thesis, Massachusetts Institute of Technology, Aug. 1989.

[7] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced (preliminary report). In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 229–239, 1988. Also appears as MIT Technical Memo MIT/LCS/TM-345.

[8] B. Bloom, S. Istrail, and A. R. Meyer. Bisimulation can't be traced. Technical Report TR 90-1150, Cornell, August 1990. (To appear in JACM).

[9] R. Burstall and J. Darlington. A transformation system for developing recursive programs,. *J. ACM*, 24(1):44–67, Jan 1977.

[10] J. Cai and R. Paige. Binding performance at language design time. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 85 – 97, Jan. 1987.

[11] J. Cai and R. Paige. Program derivation by fixed point computation,. *Science of Computer Programming*, 11(3):197–261, 1988/1989.

[12] J. Cai and R. Paige. Languages polynomial in the input plus output. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Algebraic Methodology and Software Technology*, Workshops in Computing, pages 287–302. Springer-Verlag, 1992. Conference Record of the Second AMAST.

[13] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Sci.*, 106(1):21–60, Nov. 1992.

[14] J. Carter and M. Wegman. Universal classes of hash functions,. *J. Computer and System Sci.*, 18(2):143–154, 1979.

[15] C.-H. Chang and R. Paige. From regular expressions to dfa's using compressed nfa's. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Proc. Combinatorial Pattern Matching '92*, volume 644 of *LNCS*, pages 88–108. Springer-Verlag, 1992.

[16] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench. In *Proceedings of the Workshop on Automated Verification Methods for Finite State Systems*. Springer-Verlag, 1989. LNCS 407; to appear in ACM TOPLAS.

[17] R. Cleaveland and B. Steffen. Computing behavioural relations, logically. In J. L. Albert, B. Monien, and M. R. Artalejo, editors, *Automata, Languages and Programming: 18^{th} International Colloquium*, volume 510 of *Lect. Notes in Computer Sci.*, pages 127–138. Springer-Verlag, July 1991.

[18] J. Cocke and J. Schwartz. *Programming Languages and Their Compilers*. Lecture Notes. Courant Institute, 1969.

[19] P. Cousot and R. Cousot. Constructive versions of tarski's fixed point theorems. *Pacific J. Math.*, 82(1):43–57, 1979.

[20] R. de Nicola and M. C. B. Hennessy. Testing equivalences for processes. *Theoretical Computer Sci.*, 34(2/3):83–133, 1984.

[21] R. de Simone. Higher-level synchronising devices in MEIJE-SCCS. *Theoretical Computer Sci.*, 37(3):245–267, 1985.

[22] E. Dijkstra. *A Discipline of Programming*. Prentice Hall, N.J., 1976.

[23] J. Earley. High level iterators and a method for automatically designing data structure representation. *J. of Computer Languages*, 1(4):321–342, 1976.

[24] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13:219–236, 1989/90.

[25] A. Fong and J. Ullman. Induction variables in very high level languages. In *Conference Record of the Third Annual ACM Symposium on Principles of Programming Languages*, pages 104–112, Jan. 1976.

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

[27] J. F. Groote and F. Vaandrager. Structured operational semantics and bisimulation as a congruence (extended abstract). In G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors, *Automata, Languages and Programming: 16^{th} International Colloquium*, volume 372 of *Lect. Notes in Computer Sci.*, pages 423–438. Springer-Verlag, 1989.

[28] C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, 1978.

[29] S. Johnson. Yacc - yet another compiler compiler. Technical Report Computing Science Technical Report 32, ATT Bell Laboratories, Murray Hill, N.J., 1975.

[30] K. Larsen and A. Skou. Bisimulation through probabilistic testing (preliminary report). Technical Report R 88-16, Institut for Elektroniske Systemer, Aalborg Universitetscenter, Aalborg, Danmark, June 1988.

[31] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lect. Notes in Computer Sci.* Springer-Verlag, 1980.

[32] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Sci.*, 25(3):267–310, 1983.

[33] R. Milner. *Communication and Concurrency.* Prentice Hall International Series in Computer Science. Prentice Hall, New York, 1989.

[34] L. Ong. *The lazy lambda calculus: an investigation into the foundations of functional programming.* PhD thesis, Imperial College, University of London, 1988.

[35] R. Paige. Real-time simulation of a set machine on a ram. In N. Janicki and W. Koczkodaj, editors, *Computing and Information, Vol II*, pages 69–73. Canadian Scholars' Press, May 1989.

[36] R. Paige. Efficient translation of external input in a dynamically typed language. In B. Pehrson and I. Simon, editors, *Technology and Foundations - Information Processing 94*, volume 1 of *IFIP Transactions A-51*, pages 603–608. North-Holland, Sept. 1994. Conference Record of IFIP Congress 94.

[37] R. Paige and F. Henglein. Mechanical translation of set theoretic problem specifications into efficient ram code - a case study. *Journal of Symbolic Computation*, 4(2):207–232, Aug. 1987.

[38] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. on Programming Languages and Systems*, 4(3):401–454, 1982.

[39] R. Paige and R. Tarjan. Three partition refinement algorithms. *SIAM J. Computing*, 16(6):973–989, 1987.

[40] R. Paige, R. Tarjan, and R. Bonic. A linear time solution to the single function coarsest partition problem. *Theoretical Computer Science*, 40(1):67–84, Sep. 1985.

[41] I. Phillips. Refusal testing. *Theoretical Computer Sci.*, 50:241–284, 1987.

[42] J. Schwartz, R. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, 1986.

[43] K. Snyder. The SETL2 programming language. Technical Report 490, Courant Insititute/ New York University, 1990.

[44] A. Tarski. A lattice-theoretical fixpoint theorem and its application. *Pacific J. of Mathematics*, 5:285–309, 1955.