Transformational design and implementation of a new efficient solution to the ready simulation problem

Bard Bloom\textsuperscript{a,1}, Robert Paige\textsuperscript{b,2}

\textsuperscript{a} Department of Computer Science, Cornell University, Ithaca, NY, USA
\textsuperscript{b} Department of Computer Science, New York University, New York, NY, USA

Received March 1993; revised December 1994
Communicated by M. Sintzoff

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 $O(mn^6)$ algorithm presented in Bloom (1989). 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 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 algorithm 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 supports 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 \( P + P' \), 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, \( P \Leftrightarrow P' \) 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 $P \leftrightarrow Q$, 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 $O(m \log n)$ 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

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.
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 [6-8] 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.

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 $P = \langle \Sigma_P, \rightarrow_P, i_P \rangle$, where $\Sigma_P$ 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: $p \rightarrow p'$. We omit the subscript on the transition relation $\rightarrow$ for both processes' transition relation, and write $p \Rightarrow p'$ iff $\exists p'. p \rightarrow p'$. We write $p \Rightarrow p' \in P$ iff $p \Rightarrow p'$ 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 $p \Rightarrow p'$, there is some $q'$ such that

   \[ q \Rightarrow q' \quad \text{and} \quad p' \leq q'. \]
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 \( a \),

\[ p \xrightarrow{a} \iff q \xrightarrow{a}. \]  

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 \( P \equiv Q \), if there exists a ready simulation relation \( \leq \) where \( i_p \leq i_Q \); \( P \) and \( Q \) are ready similar, \( P \equiv Q \), if \( P \equiv Q \equiv P \).

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 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, \( i_p \leq_{P,Q} i_q \iff P \equiv Q \).

Fig. 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 Fig. 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

\[ q \xrightarrow{a} q' \quad \text{and} \quad p' \leq q'. \]

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

\[ p \xrightarrow{a} p' \quad \text{and} \quad p' \leq q'. \]

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.

\(^2\) Indeed, the union of any set of ready simulation relations between arbitrary processes is a ready simulation relation.
Fig. 1. Maximum ready simulation and bisimulation relations between processes $P$ and $Q$. (a) Dotted arrows represent maximum ready simulations $\leq_{P,Q}$ and $\leq_{Q,P}$. (b) Dotted double arrows represent maximum bisimulation relations between $P$ and $Q$.

Because of this symmetry, processes $P$ and $Q$ are bisimilar ($P \leftrightarrow Q$) 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. Fig. 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 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 implementation. 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 \(\text{gfp}(x = f(x))\) to denote the greatest solution to the equation \(x = f(x)\), viz. the greatest fixed point of \(f\). \(f\) is monotone if for any \(x, y \in X\), \(f(x) \leq f(y)\) whenever \(x \leq y\).

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 \((L, \cap, 1)\), where \(L\) is a set, \(\cap\) is a binary meet (i.e., greatest lower bound) operation, and 1 is the greatest element in the partial order induced by \(\cap\): \(x \leq y\) iff \(x = x \cap y\).

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

\[
(\max Y | Y \leq f(Y)) = \text{gfp}(Y = f(Y)) = f^i(1)
\]

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 \(s_k = \text{gfp}(Y = f(Y))\) 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, $L = 2^S$ is the powerset of $S$, $\cap$ is intersection, then $\text{gfp}(Y = f(Y))$ can be computed in $O(|S|)$ iterations by the following code:

\[
Y := S; \\
\text{while } \exists x \in (Y - f(Y)) \text{ loop} \\
\quad \% \text{ execution frequency } O(|S|) \\
\quad Y := Y - \{x\} \\
\text{end loop}
\]

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. 0

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 optimization method of strength reduction [18], iteration inversion [23], and set theoretic strength reduction [25]. The essential idea is to avoid costly repeated calculations $f(x_1, \ldots, x_k)$ occurring in a program region $R$ by maintaining equality invariants of the form

\[ C = f(x_1, \ldots, x_k), \]  

where $C, x_1, \ldots, x_k$ 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 $f(x_1, \ldots, x_k)$ that are made redundant by these steps are replaced by the variable $C$. Finite differencing improves performance when the cumulative cost of repeatedly computing $f(x_1, \ldots, x_k)$ 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 $\varphi$ and $\varphi'$ be assertions in a formal language, where the free variables of $\varphi$ and $\varphi'$ are also program variables. The Hoare formula $\{\varphi\} S \{\varphi'\}$ is said to be satisfied iff every terminating execution of $S$ that satisfies $\varphi$ in its initial state satisfies $\varphi'$ 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 $C, x_1, x_2, \ldots, x_k$ be distinct program variables, and let $\phi$ represent formula

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

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

$$\{ \phi \land C = f(x_1, \ldots, x_k) \}$$

$$C := h(C, x_1, \ldots, x_k)$$

$$x_i := g(x_i)$$

$$\{ C = f(x_1, \ldots, x_k) \}$$

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:

$$\text{domain}(R) = \{ x \mid \exists y. \langle x, y \rangle \in R \}$$

$$\text{range}(R) = \{ y \mid \exists x. \langle x, y \rangle \in R \}$$

$$R \subset R' = \{ \langle x, z \rangle \mid \exists y. \langle \langle x, y \rangle \in R' \land \langle y, z \rangle \in R \} \}$$

$$R^{-1} = \{ \langle y, x \rangle : \langle x, y \rangle \in R \}$$

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

$$R[S] = \{ y : \exists x \in S. \langle x, y \rangle \in R \} = \bigcup_{x \in S} R\{x\}$$

$$\bigcup S = \bigcup_{x \in S} x$$

For example,

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

We often use this notation for the transition relations $R_\sigma = \{ \langle p, p' \rangle : p \rightarrow p' \}$. Specifically, note that:

$$p' \in R_\sigma\{p\} \text{ iff } p \sigma p' \text{ iff } p \in R_\sigma^{-1}\{p'\}$$

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:

$$R\{x\} := R\{x\} \cup \{y\} \quad \text{--element addition}$$

$$R\{x\} := R\{x\} - \{y\} \quad \text{--element deletion}$$
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 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 processes 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 $P = \langle \Sigma_P, \rightarrow_P, i_P \rangle$ and $Q = \langle \Sigma_Q, \rightarrow_Q, i_Q \rangle$ be two processes. The largest relation satisfying property (1) in our definition of ready simulation can be specified by,

$$E_1 = \max E \subseteq \Sigma_P \times \Sigma_Q \text{ such that}$$

$$(\forall \langle p, q \rangle \in E | \{ a : \exists q' | q \stackrel{a}{\rightarrow} q' \in Q \} = \{ a : \exists p' | p \stackrel{a}{\rightarrow} p' \in P \})$$

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

We calculate $E_1$ by the following method. Scan the edges in $Q$ and $P$ in linear time to obtain the sets of actions $\{ a : \exists q' | q \stackrel{a}{\rightarrow} q' \in Q \}$ for each state $q$ in $Q$, and $\{ a : \exists p' | p \stackrel{a}{\rightarrow} p' \in P \}$ 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 $\mathcal{ER}_1$ is
a compact representation of \( E_1 \); i.e.,

\[
E_1 = \bigcup \{ (B \cap \Sigma_R) \times (B \cap \Sigma_Q) : B \in ER_1 \}
\]

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.,

\[
R_a = \{ \langle r, r' \rangle \mid r \rightarrow r' \in P \lor r \rightarrow r' \in Q \}
\]

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:

\[
\max E \subseteq E_1 \text{ such that }
(\forall \langle p, q \rangle \in E, \forall a \in A, \forall \langle p, p' \rangle \in R_a, \exists \langle q, q' \rangle \in R_a | \langle p', q' \rangle \in E)
\]

(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 \((\forall x \in S | \Phi(x)) \leftrightarrow (S = \{ x \in S | \Phi(x) \})\) to obtain the following specification equivalent to (3):

\[
\max E \subseteq E_1 \text{ such that }
(\forall \langle p, q \rangle \in E, \forall a \in A, \forall \langle p, p' \rangle \in R_a, \exists \langle q, q' \rangle \in R_a | \langle p', q' \rangle \in E)
\]

(4)

We simplify specification (4) by replacing predicate \( \exists q' \in R_a \{ q \} | q' \in E \{ p' \} \) with \( R_a \{ q \} \cap E \{ p' \} \neq \emptyset \), 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,

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

Since the function

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

(5)

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

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

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

This leads to an \( n^2 \) bound on the sizes of any binary relation defined on \( \Sigma_R \cup \Sigma_Q \), 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 $F'(E_1)$ for some integer $t$. However, a high degree of redundancy makes this approach too inefficient. Fortunately, we can use Corollary 3.4 to compute $\text{gfp}(E = F(E))$ more efficiently by performing the code shown in Fig. 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 Fig. 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 $(p_*, q_*)$ 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 Fig. 2.

5.2. Finite differencing

In this section we will recognize computational bottlenecks appearing in Fig. 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

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

$$E := E_1$$

while $\exists \langle p_*, q_* \rangle \in (E - F(E))$ loop

% execution frequency $O(n^2)$

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

end loop

Fig. 2. First approximation to a ready simulation algorithm.
which can be rewritten more conveniently,
\[
\{ (p, q) \in E \mid (\exists a \in A \mid R_a \{p\} - (E^{-1} \lhd R_a) \{q\} \neq \emptyset) \}
\]
where
\[
E^{-1} \lhd R_a = \{ (qq, pp) : \exists w. (\langle qq, w \rangle \in R_a \land pp \in E^{-1} \{w\}) \}
\]
according to the set-theoretic definition of relation composition given earlier. In order to maintain the invariant
\[
W = \{ (p, q) \in E \mid (\exists a \in A \mid R_a \{p\} - (E^{-1} \lhd R_a) \{q\} \neq \emptyset) \}
\]
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
\[
T_1(a) = E^{-1} \lhd R_a
\]
for each \(a \in A\). For fixed \(a \in A\) the preceding relational composition evaluates to the set of pairs \(\langle qq, pp \rangle\)

\[
\begin{align*}
qq & \vdash R_a \\
\vdots & \\
pp & \ldots \rightarrow (q)
\end{align*}
\]
for which there exists a \(q\) such that \(\langle qq, q \rangle\) belongs to \(R_a\) and \(\langle pp, q \rangle\) belongs to \(E\).

Assume that for each \(a \in A\) invariant \(T_1(a)\) holds on entry to the while-loop of Fig. 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 Fig. 3 just before \(E\) is updated.

\begin{verbatim}
for aa \in A such that R^{-1}_{aa} \{q_*\} \neq \emptyset loop
    for qq_* \in R^{-1}_{aa} \{q_*\} loop
        % execution frequency O(mn)
        if \(|R_{aa} \{qq_*\} \cap E \{p_*\}| = 1\) then
            T1(aa) \{qq_*\} := T1(aa) \{qq_*\} - \{p_*\}
        end if
    end loop
end loop
\end{verbatim}

Fig. 3. Updating \(T_1\).
The innermost for-loop occurring in Fig. 3 follows from the Finite Difference Schema of Lemma 3.6 using the differential identity

\[(E \setminus \{\langle p_*, q_* \rangle \})^{-1} \subseteq R_a\]

\[= E^{-1} \subseteq R_a - \{\langle qq, p_* \rangle : qq \in R_a^{-1}\{q_*\} \mid |R_a\{qq\} \cap E\{p_*\}| = 1\}\]

See Appendix 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 \(E^{-1} \subseteq R_a\) 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 Fig. 3. Since each pair \(\langle p_*, q_* \rangle\) of values can be selected only once at the top of the while-loop in Fig. 2, then the number of times that \(R_a\{q_*\}\) can be searched for each distinct pair of values \(q_*\) and \(aa\) is at most \(n\) in Fig. 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 Fig. 3 too inefficient. These two problems are overcome by also maintaining the two invariants,

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

and

\[T3 = \{\langle q, a \rangle : a \in A, q \in \text{range}(R_a)\} .\]

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 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 Fig. 4 instead of the code in Fig. 3. The code to update \(T2(aa)\)

```
for aa ∈ T3\{q_*\} loop
  % execution frequency \(O(mn)\)
  for qq_* ∈ R_{aa}^{-1}\{q_*\} loop
    % execution frequency \(O(mn)\)
    if |T2(aa)\{\langle p_*, qq_* \rangle \}| = 1 then
      T1(aa)\{qq_*\} := T1(aa)\{qq_*\} - \{p_*\}
    end if
    T2(aa)\{\langle p_*, qq_* \rangle \} := T2(aa)\{\langle p_*, qq_* \rangle \} - \{q_*\}
  end loop
end loop
```

Fig. 4. Maintaining invariants for \(T1\) and \(T2\).
for \( pp_* \in R_{aa}^{-1}\{p_*\} \) loop
% execution frequency \( O(mn) \)
if \( \langle pp_*, qq_* \rangle \in (E - W) \) then
\[ W := W \cup \{ \langle pp_*, qq_* \rangle \} \]
end if
end loop

Fig. 5. Updating \( W \).

follows from the Finite Difference Schema Lemma 3.6 using the differential identity,
\[
\{ \langle p, qq \rangle : \langle qq, q \rangle \in R_{aa}, p \in (E - \{ \langle p_*, q_* \rangle \})^{-1}\{q\} \}
\]
\[
= \{ \langle p, qq \rangle : \langle qq, q \rangle \in R_{aa}, p \in E^{-1}\{q\} \} - \{ \langle p_*, qq \rangle : q \in R_{aa}^{-1}\{q_*\} \}
\]

Now we can consider how to maintain the top-level invariant,
\[ W = \{ (p, q) \in E | (\exists a \in A | R_a\{p\} - T1\{q\} \neq \emptyset) \}
\]
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
\[ T1(a)\{qq_*\} := T1(a)\{qq_*\} - \{p_*\} \]

we can update \( W \) by executing the code shown in Fig. 5, which can be made more efficient by maintaining the additional invariant,
\[ WC = E - W \]

Since the predicate \( |T2(aa)\{p_*, q_*\}| = 1 \), appearing in Fig. 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 Fig. 5 is at most \( O(mn) \).

The update for \( W \) relative to the modifications to \( T1(aa)\{qq_*\} \) follows from the Finite Difference Schema and the differential identity,
\[
\{ \langle p, q \rangle \in E \mid (\exists a \in A | R_a\{p\} - T1\{q\} \neq \emptyset) \} - \{ \langle p, q \rangle \in E | (\exists a \in A | R_a\{p\} - T1\{q\} \neq \emptyset) \}
\]
\[
\cup \{ \langle p, qq_* \rangle : p \in R_{aa}^{-1}\{p_*\} | \langle p, qq_* \rangle \in E \land (\exists a \in A | R_a\{p\} - T1\{qq_*\} \neq \emptyset) \}
\]

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

Just prior to the change
\[ T1(aa)\{qq_*\} := T1(aa)\{qq_*\} - \{p_*\} \]

we can update \( W \) by executing the code shown in Fig. 5, which can be made more efficient by maintaining the additional invariant,
which is obtained as follows. First we calculate,

\[
\left( R_a(p) - \begin{cases} 
  & \text{if } q = q q_\ast \land aa = a \\
  \quad \text{then } & T I(a)\{q q_\ast\} - \{p_\ast\} \\
  \quad \text{else } & T I(a)\{q\} 
\end{cases} \right) \neq \emptyset
\]

\[
= \left( \begin{cases} 
  & \text{if } q = q q_\ast \land aa = a \\
  \quad \text{then } & (R_a(p) - (T I(a)\{q\} - \{p_\ast\}) \neq \emptyset) \\
  \quad \text{else } & (R_a(p) - T I(a)\{q\} \neq \emptyset) 
\end{cases} \right)
\]

\[
= \left( \begin{cases} 
  & \text{if } q = q q_\ast \land aa = a \\
  \quad \text{then } & ((R_a(p) - T I(a)\{q\}) \cup (R_a(p) \cap \{p_\ast\}) \neq \emptyset) \\
  \quad \text{else } & (R_a(p) - T I(a)\{q\} \neq \emptyset) 
\end{cases} \right)
\]

\[
= (R_a(p) - T I(a)\{q\} \neq \emptyset) \lor (a = aa \land q = q q_\ast \land p \in R_a^{-1}\{p_\ast\})
\]

The rest of the derivation proceeds as follows.

\[
\left\{ \langle p, q \rangle \in E \mid \exists a \in A \left( \begin{cases} 
  & \text{if } q = q q_\ast \land aa = a \\
  \quad \text{then } & R_a(p) - (T I(a)\{q q_\ast\} - \{p_\ast\}) \neq \emptyset \\
  \quad \text{else } & T I(a)\{q\} 
\end{cases} \right) \right\}
\]

\[
= \left\{ \langle p, q \rangle \in E \mid \exists a \in A \left( R_a(p) \\
  \quad - T I(a)\{q\} \neq \emptyset \lor (a = aa \land q = q q_\ast \land p \in R_a^{-1}\{p_\ast\}) \right) \right\}
\]

\[
= \left\{ \langle p, q \rangle \in E \mid \exists a \in A \left( R_a(p) \\
  \quad - T I(a)\{q\} \neq \emptyset \lor (a = aa \land q = q q_\ast \land p \in R_a^{-1}\{p_\ast\}) \right) \right\}
\]

\[
= \left\{ \langle p, q \rangle \in E \mid \exists a \in A \left( R_a(p) - T I(a)\{q\} \neq \emptyset \right) \right\}
\]

\[
\cup \left\{ \langle p, q q_\ast \rangle : p \in R_a^{-1}\{p_\ast\} \mid \langle p, q q_\ast \rangle \in E \land \exists a \in A \left( R_a(p) - T I(a)\{q q_\ast\} \neq \emptyset \right) \right\}
\]
Observe that invariant $WC$ and $W$ can be maintained collectively by executing the following code just before the modification to $TI$.

```plaintext
for $pp_*, qq_* \in R_{aa}^{-1}\{p_*, q_*\}$ loop
  % execution frequency $O(mn)$
  if $\langle pp_*, qq_* \rangle \in WC$ then
    % execution frequency $O(n^2)$
    $WC := WC - \{\langle pp_*, qq_* \rangle\}$
    $W := W \cup \{\langle pp_*, qq_* \rangle\}$
  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 $TI$ is never used, that a set of reference counts

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

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 Fig. 6.

To make an initial performance analysis of the code in Fig. 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

```plaintext
while $\exists \langle p_*, q_* \rangle \in W$ loop
  for $aa \in T3\{q_*\}$ loop
    for $qq_* \in R_{aa}^{-1}\{q_*\}$ loop
      if $T2count(aa)(p_*, qq_*) = 1$ then
        for $pp_* \in R_{aa}^{-1}\{p_*\}$ loop
          if $\langle pp_*, qq_* \rangle \in WC$ then
            $WC := WC - \{\langle pp_*, qq_* \rangle\}$
            $W := W \cup \{\langle pp_*, qq_* \rangle\}$
          end if
        end loop
      end if
    end loop
  end loop
end if
$T2count(aa)(p_*, qq_*) := T2count(aa)(p_*, qq_*) - 1$
end loop
$W := W \cup \{\langle p_*, q_* \rangle\}$
end loop
% $WC$ is the desired ready simulation relation.
```

Fig. 6. Prototype code.
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 bounds depend entirely on the total number of while-loop iterations and the total number of for-loop iterations through \( R_a^{-1}\{q^*_a\} \) and through \( R_a^{-1}\{p^*_a\} \). By previous analysis, the worst case time for the code in Fig. 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 \( T_1, T_2\text{count}, T_3, W, \) 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 \( T_3 = \{ (q, q) : a \in A, q \in \text{range}(R_a) \} \) by executing,

\[
T_3 := \emptyset \\
\text{for } a \in A \text{ loop} \\
\text{% execution frequency } O(|A|) = O(m) \\
\text{for } q^*_a \in \text{domain}(R_a^{-1}) \text{ loop} \\
\text{% execution frequency } O(m) \\
T_3\{q^*_a\} := T_3\{q^*_a\} \cup \{a\} \\
\text{end loop} \\
\text{end loop}
\]

Subsequently, we can establish \( T_1(a) = E^{-1} a R_a \) and \( T_2\text{count} \) using a search through \( T_3 \) according to the code in Fig. 7. This code is obtained by reducing the problem of establishing invariants \( T_1 \) and \( T_2\text{count} \) to the problem of maintaining them, first with respect to an assignment \( E := \emptyset \), and then with respect to successively adding each pair \( \langle p^*_a, q^*_a \rangle \) 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 Fig. 4. Arguments to support the \( O(mn) \) bound on the frequencies of execution shown in Fig. 7 are dual to those previously given for the code found in Fig. 4.

In order to establish

\[
W = \{ \langle p, q \rangle \in E \mid (\exists a \in A \mid R_a\{p\} - T_1(a)\{q\} \neq \emptyset) \},
\]

we will make use of the three new invariants,

\[
w\text{count}_2(a)(p, q) = |R_a\{p\} - T_1(a)\{q\}|
\]

for all \( a \in A \) and \( \langle p, q \rangle \in E \) such that \( R_a\{p\} \neq \emptyset \),

\[
T_4 = \{ \langle p, a \rangle : a \in A, p \in \text{domain}(R_a) \}
\]
\[ T_2 \text{count} := \emptyset \]
\[ T_1 := \emptyset \]
for \( \langle p_*, q_* \rangle \in E \), loop
\% execution frequency \( O(n^2) \)
for \( a \in T_3 \{ q_* \} \), loop
\% execution frequency \( O(mn) \)
for \( qq_* \in R_*^{-1} \{ q_* \} \), loop
\% execution frequency \( O(mn) \)
\% assume \( T_2 \text{count}(a)(p_*, qq_*) = 0 \) if undefined
if \( T_2 \text{count}(a)(p_*, qq_*) = 0 \) then
\[ T_1(a) \{ qq_* \} := T_1(a) \{ qq_* \} \cup \{ p_* \} \]
end if
\[ T_2 \text{count}(a)(p_*, qq_*) := T_2 \text{count}(a)(p_*, qq_*) + 1 \]
end loop
end loop
end loop

Fig. 7. Code to establish \( T_1 \) and \( T_2 \text{count} \).

and
\[ \text{wcoun}t(p, q) = \sum_{a \in T_4 \{ p \}} \text{wcoun}t_2(a)(p, q) \]

Based on these invariants, we see that
\[ W = \{ \langle p, q \rangle \in E | (\exists a \in A | R_a \{ p \} - T_1(a) \{ q \} \neq \emptyset) \} \]
\[ = \{ \langle p, q \rangle \in E | (\exists a \in A | \text{wcoun}t_2(a)(p, q) > 0) \} \]
\[ = \{ \langle p, q \rangle \in E | \text{wcoun}t(p, q) > 0 \} \]

We also know that for all \( a \in A \), \([R_a \{ p \} \neq \emptyset \) iff \( a \in T_4 \{ p \} \).

In order to establish these invariants, we first recognize that \( T_4 \) can be established efficiently together with \( T_3 \); the code is given in Fig. 8. The computation to establish the remaining invariants is broken into two parts for convenience. If we assume that \( T_1(a) = \emptyset \) for all \( a \in A \), then
\[ W = \{ \langle p, q \rangle \in E | (\exists a \in A | R_a \{ p \} - T_1(a) \{ q \} \neq \emptyset) \} \]
\[ = \{ \langle p, q \rangle \in E | (\exists a \in A | R_a \{ p \} \neq \emptyset) \} \]
\[ = \{ \langle p, q \rangle \in E | T_4 \{ p \} \neq \emptyset \} \]

and
\[ \text{wcoun}t_2(a)(p, q) = |R_a \{ p \}| \]

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

Using the preceding identities we can establish initial approximations of invariants \( W \), \( WC \), \( \text{wcoun}t_2 \), and \( \text{wcoun}t \) by the code in Fig. 9. This code arises by first
Fig. 8. Code to establish $T_3$ and $T_4$.

```plaintext
$T_3 := \emptyset$
$T_4 := \emptyset$

for $a \in A$ loop
  for $q_* \in \text{range}(R_a)$ loop
    \%
    \% execution frequency $O(m)$
    $T_3\{q_*\} := T_3\{q_*\} \cup \{a\}$
  end loop
  for $q_* \in \text{domain}(R_a)$ loop
    \%
    \% execution frequency $O(m)$
    $T_4\{q_*\} := T_4\{q_*\} \cup \{a\}$
  end loop
end loop
```

Fig. 9. Establishing $W$, $WC$, $wcount$ and $wcount2$ ignoring $T_1$.

```plaintext
wcount := \emptyset; wcount2 := \emptyset
for $a \in A$ loop
  wcount2(a) := \emptyset
end loop
$W := \emptyset$
$WC := E_1$
for $\langle p_*, q_* \rangle \in E_1$ loop
  if $T_4\{p_*\} \neq \emptyset$ then
    $WC := WC - \{\langle p_*, q_* \rangle\}$
    $W := W \cup \{\langle p_*, q_* \rangle\}$
    for $a \in T_4\{p_*\}$ 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
```

maintaining these invariants (under the assumption that $T_1$ is empty) with respect to assignment $E := \emptyset$, 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 $T_1$ into account. This code, which is given in Fig. 10, arises from maintaining the three invariants relative to imaginary operations of the form $T_1(a)\{qq_*\} := T_1(a)\{qq_*\} \cup \{p_*\}$ that
augment $T1$ (assumed to be empty initially) repeatedly until it reaches its correct value. The execution frequency of the inner for-loop within Fig. 10 is justified in a way dual to the argument supporting the execution frequency appearing in Fig. 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 $T1(a)^{-1} = \text{domain}(T2\ count(a))$ (so that $T1(a)$ can be eliminated), that $W$ and domain $\text{wcount}$ can be equivalenced, and that the first loop above (which is used to partially establish $W$, $WC$, $\text{wcount2}(a)$, and $\text{wcount}$) can be jammed into the same loop that establishes $T1(a)$ and $T2\ count$. Consequently, we can establish all the invariants needed within the main while-loop shown in Fig. 6 by executing the code in Fig. 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 $\text{domain}(\text{wcount2}(a))$, $\text{domain}(T2\ count(a))$, $\text{domain}(R_n^{-1})$, $WC$, $\text{domain}(\text{wcount})$, $\text{domain}(T4)$, and $\text{domain}(T3)$. We also need to support searching through a set or through the domain of a map in 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
T3 := \emptyset; T4 := \emptyset; wcount := \emptyset; wcount2 := \emptyset; T2 count := \emptyset
for a \in A loop
  wcount2(a) := \emptyset
  T2 count(a) := \emptyset
  for q* \in range(R_a) do T3(q*) := T3(q*) \cup \{a\}
  for q* \in domain(R_a) do T4(q*) := T4(q*) \cup \{a\}
end loop
WC := F, % pointer assignment can be used for efficiency
for \langle p*, q* \rangle \in WC loop
  for a \in T3(q*) loop
    % assume T2 count(a)(p*, q*) = 0 if undefined
    for qq* \in R^{-1}_a(q*) do T2 count(a)(p*, qq*) := T2 count(a)(p*, qq*) + 1
  end loop
  if T4(p*) \neq \emptyset then
    WC := WC - \{\langle p*, q* \rangle\}
    for a \in 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 \in A loop
  for \langle p*, qq* \rangle \in domain T2 count(a) loop
    for pp* \in R^{-1}_a(p*) loop
      if wcount2(a)(pp*, qq*) > 0 then
        if wcount(pp*, qq*) = 1 do WC := WC \{ \langle pp*, qq* \rangle\}
        % 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

Fig. 11. Full initialization.

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 \( \Sigma_P \) and \( \Sigma_Q \). That is, we assume unique integer identifiers from 1 to \( |\Sigma_P| \) for the states in \( \Sigma_P \), and unique integer identifiers from 1 to \( |\Sigma_Q| \) for the states in \( \Sigma_Q \). 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 Fig. 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 $\Sigma_P$ 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 $R_a^+(i)$, where set $R_a^-(i)$ is a one-way list of integers designating $\Sigma_P$-states. For each $\Sigma_Q$ state $i$, component $Q_a(i)$ is a record with only one field - a pointer to $R_a^+(i)$, where set $R_a^+(i)$ is a one-way list of integers designating $\Sigma_Q$-states. Hence, these pre-images can be searched in time proportional to their cardinalities.

Array $PQ_a$ is used to store $wcount_2(a)$ and $T2count(a)$. For each $\Sigma_P$-state $i$ and $\Sigma_Q$-state $j$, $PQ_a(i,j)$ is a record with five fields. One of these fields stores $T2count(a)(i,j)$, and another stores $wcount_2(a)(i,j)$. 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. Fig. 11). Within this loop the two cursor fields (which represent states $p^*$ and $q^*$) 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 $PQ_a$.

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 $\Sigma_P$ 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 access array \( M \). Similarly, for each \( \Sigma_q \) 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 \( \Sigma_p \)-state \( i \) and \( \Sigma_q \)-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 \( wcounl(i,j) \). 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 Fig. 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. Fig. 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. Fig. 6). Array access is needed to obtain a unit-time membership test inside the while-loop.

Implementation of \( W \) together with \( wcounl \) (recall that \( W \) and domain \( wcounl \) 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 Fig. 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 \( wcount(\mathbf{pp}_*, \mathbf{qq}_*) := \)
\( wcount(\mathbf{pp}_*, \mathbf{qq}_*) - 1 \), appearing in the last for-loop of the initialization (cf. Fig. 11),
resets the count \( wcount(\mathbf{pp}_*, \mathbf{qq}_*) \) 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 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 implemen-
tation 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.

Appendix A. Sample correctness proof

In this section, we present a detailed proof that the code in Fig. 3 is correct. We
state our conditions in Hoare logic, though our reasoning is largely semantic.
Let
\[ \Phi = \forall a \in A \mid Tl(a) = E^{-1} \Rightarrow R_a \]
be the condition that \( Tl \) has the correct value. In Hoare logic, we wish to show that \( \Phi \) is an invariant:
\[ \{ \Phi \} \text{Fig. 3; } E := E - \{ \langle p_*, q_* \rangle \} \{ \Phi \} \]

### A.1. Eliminating loops

In Fig. 3, we repeatedly use for loops our 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 nonlooping, 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 \( \varphi \) and \( \varphi' \), \( \{ \varphi \} S \{ \varphi' \} \) 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

\[
\text{for } x \text{ in } Q \text{ loop} \\
\text{if } P(x, y) \text{ then} \\
T(x) := T \{ x \} - y \\
\text{end if} \\
\text{end loop}
\]

is Hoare equivalent to the program

\[ T := T - \{ \langle x, y \rangle \mid x \in Q(x) \land P(x, y) \} \]

(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. \( \Box \)

**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 \( T, x \) and \( x \) do not appear in \( Q \) or \( e \), then the program

\[
\text{for } x \text{ in } Q \text{ loop} \\
T(x) := T(x) - e(x) \\
\text{end loop}
\]
is Hoare equivalent to the program

\[ T := \{ \langle x, T(x) \rangle \mid x \in \text{domain}(T) \land x \notin Q \} \]

\[ \cup \{ \langle x, T(x) - e(x) \rangle \mid x \in \text{domain}(T) \land x \in Q \} \]

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

\[
\text{for } a \text{ in } A \text{ such that } R_a^{-1}\{q_*\} \neq \emptyset \text{ loop}
\]

\[
\text{for } q_q^* \text{ in } R_a^{-1}\{q_*\} \text{ loop}
\]

\[
\text{if } |R_a\{q_q^*\} \cap E\{p_*\}| = 1 \text{ then}
\]

\[
T_1(a)\{q_q^*\} := T_1(a)\{q_q^*\} - \{p_*\}
\]

\[
\text{end if}
\]

\[
\text{end loop}
\]

\[
\text{end loop}
\]

by the simpler Hoare equivalent code,

\[
\text{for } a \text{ in } A \text{ such that } R_a^{-1}\{q_*\} \neq \emptyset \text{ loop}
\]

\[
T_1(a) := T_1(a) - \{ \langle q_q, p_* \rangle \mid q_q \in R_a^{-1}\{q_*\} \land |R_a\{q_q\} \cap E\{p_*\}| = 1 \}
\]

\[
\text{end loop}
\]

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

\[
\text{for } a \text{ in } A \text{ loop}
\]

\[
T_1(a) := T_1(a) - \{ \langle q_q, p_* \rangle \mid q_q \in R_a^{-1}\{q_*\} \land |R_a\{q_q\} \cap E\{p_*\}| = 1 \}
\]

\[
\text{end loop}
\]

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

\[
T_1 := \{ \langle a, T_1(a) \rangle \mid a \in \text{domain}(T_1) \land a \notin A \}
\]

\[ \cup \{ \langle a, T_1(a) \rangle - \{ \langle q_q, p_* \rangle \mid q_q \in R_a^{-1}\{q_*\} \land |R_a\{q_q\} \cap E\{p_*\}| = 1 \} \} \mid a \in \text{domain}(T_1) \land a \in A \}
\]

Finally, since \( A = \text{domain}(T_1) \), we can simplify the code just above to the single assignment,

\[
T_1 := \{ \langle a, T_1(a) \rangle - \{ \langle q_q, p_* \rangle \mid q_q \in R_a^{-1}\{q_*\} \land |R_a\{q_q\} \cap E\{p_*\}| = 1 \} \} \mid a \in A \}
\]

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:

\[
k = 1
\]

\[
C = T_1
\]
$x_1 = E,$

$$f(E) = \{ \langle a, E^{-1} \triangleleft R_a \rangle | a \in A \}$$

$$g(E) = E - \{ \langle p_*, q_* \rangle \}$$

$$h(T_1, E) = \{ \langle a, T_1(a) - \{ \langle qq, p_* \rangle | qq \in R_a^{-1} \{ q_* \} \} \}
\wedge (|R_a\{qq\} \cap E\{p_*\}| = 1) \rangle | a \in A \}$$

$$\varphi = f(g(E)) = h(f(E), E)$$

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

$$\{ \langle a, (E - \{ \langle p_*, q_* \rangle \})^{-1} \triangleleft R_a \rangle | a \in A \}$$

$$= \{ \langle a, E^{-1} \triangleleft R_a - \{ \langle qq, p_* \rangle | qq \in R_a^{-1} \{ q_* \} \} \wedge |R_a\{qq\} \cap E\{p_*\}| = 1 \rangle | a \in A \}$$

which is true iff these two functions are pointwise equal; that is, for all $a$ in $A$, $$(E - \{ \langle p_*, q_* \rangle \})^{-1} \triangleleft R_a = E^{-1} \triangleleft R_a - \{ \langle qq, p_* \rangle | qq \in R_a^{-1} \{ q_* \} \} \wedge |R_a\{qq\} \cap E\{p_*\}| = 1$$

We show this by calculation; i.e.,

$$(E - \{ \langle p_*, q_* \rangle \})^{-1} \triangleleft R_a$$

$$= \{ \langle q, pp \rangle | \exists qq (\langle q, qq \rangle \in R_a \wedge \langle qq, pp \rangle \in (E - \{ \langle p_*, q_* \rangle \})^{-1}) \}$$

$$= \{ \langle q, pp \rangle | R_a\{q\} \cap (E - \{ \langle p_*, q_* \rangle \})\{pp\} \neq \emptyset \}$$

$$= \{ \langle q, pp \rangle | (R_a\{q\} \cap E\{pp\}) - (R_a\{q\} \cap (\{ \langle p_*, q_* \rangle \}\{pp\})) \neq \emptyset \}$$

$$= \{ \langle q, pp \rangle | (R_a\{q\} \cap E\{pp\}) \neq \emptyset \wedge (pp = p_* \wedge q_* \in R_a\{q\})$$

$$\Rightarrow |R_a\{q\} \cap E\{pp\}| \neq 1) \}$$

$$= \{ \langle q, pp \rangle | R_a\{q\} \cap E\{pp\} \neq \emptyset \wedge \neg (pp = p_* \wedge q_* \in R_a\{q\})$$

$$\wedge |R_a\{q\} \cap E\{pp\}| = 1) \}$$

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

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

Appendix 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
Acknowledgements

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

References