Reliable Computing 1 (1) (1995), pp. 65–75

An informal introduction to a high level language with applications to interval mathematics

DANIEL E. COOKE

The main problem of interval computations is as follows: *given* sets of possible values X_i for variables x_i, and an algorithm $f : R^n \to R$, to *estimate* the range $f(X_1, \ldots, X_n)$ of the possible values of $f(x_1, \ldots, x_n)$. In many real-life situations, sets X_i are not intervals. To handle such problems, it is desirable to add set data type and operations with sets to a programming language. It is well known that the entire mathematics can be formulated in terms of sets. So, if we already have a set as a data type, why have anything else? The main reason is that expression in terms of sets is often clumsy. To avoid this clumsiness, it has been suggested to use not only sets, but also *bags* (*multisets*), in which an element can have multiple occurrences. Bags are used in many areas of Computer Science, and recently, several languages have appeared that use the bag as a basic data type.

In this paper, we explain the main ideas behind bag languages, and we also show:

- that bag languages are naturally parallelizable, thus leading to a parallelization of the coresponding generalized interval computations;
- and that bag languages can be also helpfully applied to traditional interval computations (where sets X_i are intervals).

Неформальное введение в язык высокого уровня с приложениями к интервальной математике

Д. Е. Кук

Основная задача интервальных вычислений формулируется следующим образом: даны множества возможных значений X_i переменных x_i и алгоритм $f : R^n \to R$; требуется оценить множество $f(X_1, \ldots, X_n)$ возможных значений функции $f(x_1, \ldots, x_n)$. На практике множества X_i часто не являются интервалами. Чтобы справиться с такими задачами, желательно добавить множества как тип данных и операции с множествами в языки программирования. Известно, что вся математика может быть изложена в терминах множеств. Возникает вопрос: если у нас есть множество как тип данных, зачем нужно что-то еще? Основное возражение заключается в том, что математические конструкции из множеств часто очень громоздки. Чтобы избежать этого, предложено использовать не только множества, но и *мультимножества* (bags), в которые один и тот же элемент может входить по нескольку раз. Мультимножества используются во многих областях информатики, и в последнее время появилось несколько языков программирования, в которых мультимножества являются основным типом данных.

В настоящей работе излагаются основные концепции языков, использующих мультимножества, а также показывается, что:

- языки с мультимножественным типом данных естественно параллелизуются, в результате чего соответствующие обобщенные интервальные вычисления также приобретают параллельный вид;
- мультимножества и использующие их языки выгодно применять и для обычных интервальных вычислений (в которых множества X_i являются интервалами).

1. Interval computations are computations with sets, so it is desirable to have set operations in the programming language

1.1. Usually, intervals represent our uncertainty

In many real-life situations, we do not know the precise value of a physical quantity. Usually, possible values form an interval. For example, if we measure voltage V with precision ε, and the measurement result is equal to $\tilde{V}$, this means that the actual voltage belongs to an interval $[\tilde{V} - \varepsilon, \tilde{V} + \varepsilon]$.

If we have several independent physical quantities $x_1, \ldots, x_n$, and we know an interval X_i of possible values of each of them, then the set of all possible values of the vector $\vec{x} = (x_1, \ldots, x_n)$ is just a set $X_1 \times X_2 \times \cdots \times X_n$ of all possible tuples $(x_1, \ldots, x_n)$ with $x_i \in X_i$.

1.2. In some cases, non-interval sets are necessary

In some cases, the set X of possible values of a physical quantity x is not an interval. For example, we may know that x takes only integer values, but we are not sure what the value is, so there are several possible values. Such situations occur in quantum physics, where many quantities (spin, angular momentum, charges, etc) can take only the values from some discrete set (that is called a *spectrum* of this quantity). In this case, if we know the approximate value of x, and know the precision, then the possible values of x form a finite set (usually, an ordered finite set).

Another situation when intervals are not suficient is when we have several physical quantities $x_1, \ldots, x_n$ that are *dependent* in the sense that some apriori relation must be known (e.g., $x_1 \geq x_2$). In this case, the set of all possible values of $\vec{x}$ may be different from the Cartesian product of intervals

1.3. Interval computations

Main problem. One of the main objectives of interval mathematics is to analyze the following situation:

Suppose that:

- *We know* the sets of possible values $X_1, \ldots, X_n$ for several quantities $x_1, \ldots, x_n$,
- and *we know* that some other quantity is related to x_i by a formula $y = f(x_1, \ldots, x_n)$.

If we take different values $x_i \in X_i$, we will end up with different values of y.

The problem: to describe the set Y of possible values of y.

Of course, for *every* such *situation*, we can write *a special program* that computes this set Y, without inventing any new formalism. It is desirable, however, to have a *software tool* that, given an expression for $f(x)$, and the description of X, would generate the set Y.

The idea of interval computations. Such a tool (*naive interval computations*) exists for the case when X_i are intervals, and is based on the following idea:

The algorithm that computes $f(x)$ can be represented as a sequence of elementary arithmetic operations ($+$, $-$, $*$, $/$, etc). As results of these steps, we get the values $r_1, r_2, \ldots, r_n$ (where n is the total number of computational steps).

Example. For $f(x) = x - x^2$ we have:

- $r_1 = x * x$; and
- $f(x) = r_2 = x - r_1$.

The idea of naive interval computations is to apply the same operations, in the same order, but to sets, and not to individual numbers. In order to do that, we must extend operations with numbers to operations with sets:

$$X * Y = \{x * y \mid x \in X, y \in Y\}. \tag{1}$$

Example. In the above example, if $X = [0, 1]$, we have:

- $R_1 = [0, 1] * [0, 1] = [0, 1]$;
- $f(X) = R_2 = X - R_1 = [0, 1] - [0, 1] = [-1, 1]$.

Naive interval computations usually overestimate $f(X)$, so to make better estimates, we must change the original algorithm (e.g., apply *centered form*).

1.4. A natural generalization: operations with sets

In case our uncertainty about the values of x_i is represented by arbitrary sets (not necessarily intervals), a reasonable idea is to apply the same method, but with operations (1) defined for arbitrary sets.

Therefore, it is desirable to have a programming language that would:

- include sets as a data type;
- include operations with sets.

2. Sets are a natural specification language

We have argued that programming language must, among other data types, include sets. But if we have such a powerful data type as sets, do we need anything else? To find that out, let's consider the very idea of a specification language.

2.1. Specification languages: one of the main tools of Software Engineering

One of the main goals of Software Engineering is to provide support in writing programs. Before we can write a program, we must know what this program is supposed to do. In other words, we must know the *specification* for this program. Customers who order these programs

formulate these specifications in a natural language. The task of the programmer is to translate these specifications into an actual program, written in the required programming language.

Part of this translation task is more of an art (i.e., successful parallelization of an algorithm is still mainly an art), but a large part of the programmer's work is more or less routine. One of the main difficulties which make the programmers' work non-trivial is that the gap between natural language and the majority of programming languages is too wide:

1) natural language is *ambiguous*, while programming languages are very precise;

2) natural language is often *non-algorithmic*: we just say "solve an equation" without specifying how to solve it, while the program must either contain a step-by-step description, or it must at least lead to such a description.

To simplify a programmer's task, it is natural to try to cover this gap in two steps:

1) to *translate* the informal (and sometimes ambiguous) natural-language description *into a* description in some *formal language*;

2) to *transform this formal description into* an actual *program*.

This intermediate formal language is called a *specification language*, and the corresponding formal description of the task is called a *specification for a program*.

Remark. Since a specification language is formal, it becomes possible to develop algorithms that transform (some) specifications into actual programs. In other words, for some classes of specifications, it becomes possible to *automate* programming. When such automation becomes possible for the entire specification language, this language becomes a *programming language*, because we can simply write a specification in this language and let the compiler translate it into executable code.

Being, so to say, "elevated" to the status of a programming language often does not prevent a language from being a useful specification language as well.

Let's give two examples:

- *Pascal* is a good pedagogical language, but it is rarely used in real-life applications. However, it is a very widely used specification language: Pascal-style pseudocode is often used in Computer Science journals and books to describe algorithms.
- *Prolog* and its modifications are an excellent and a widely accepted way to describe knowledge. However, due to the fact that Prolog (at least in its existing implementations) is not very fast, real knowledge-based systems often use LISP or other faster languages.

2.2. Sets are a natural specification language

Crudely speaking, there are two tendencies in designing specification languages:

- some of these languages are closer to natural language (like Prolog);
- some of them are closer to the existing programming languages.

Languages of the second type are easier to write, easier to compile, but they are further away from the ideal and thus, put an unnecessary burden on the specification-writer. The more challenging task (but at the same time more rewarding) is to write specification languages that are closer to the natural language.

The task of designing such a language may seem practically impossible (has not Artificial Intelligence been trying to do that for several decades already?), if we do not recall that mathematics has been doing exactly that for several millennia. All formalized models that have been developed during that time are formulated in the language of mathematics. The language of mathematics is based on the notion of a set (i.e., on the formalization of our usage of the words "belongs to", "is an element of", etc).

There are many notions in modern mathematics that are much more complicated than the notion of a set, but they can all be described in terms of sets and their theory (*set theory*) (e.g., a function $f : X \to Y$ is defined in mathematics as a set $\{(x, f(x)) \mid x \in X\}$ of pairs $(x, f(x))$ for all $x \in X$. Anyone who has gone through a course of Discrete or Computer Math remembers that such descriptions are indeed possible.

So, a natural idea is to use sets as a basic datatype for a specification language. This idea has been implemented by J. Schwartz (see, e.g., [17]) in his Set Language (SETL for short). Moreover, it has been proven that there exists an algorithm that transforms every specification from SETL into working code. In this sense, SETL is also a programming language. It even has some applications as a programming language, although not so many as it would seem from our description of Set Language as a natural specification language. Why?

3. Drawbacks of set languages. Bags

3.1. The main drawback of sets

The main drawback of set languages is that although all mathematical practice can be expressed *in terms of sets*, even for the simplest notions, *this expression is* often clumsy and too *complicated* (again, anyone who has gone through a course on Discrete or Computer Math can confirm that). For example, expressing the notion of a *bag* in terms of sets, is clumsy.

To understand what a bag is let us recall what a set is: a set is a collections of elements (over some domain). So, a natural idea is to describe, e.g., collections of records in a database as sets. For example, if we want to describe the last names of all the students from our Department, it seems reasonable to use a set of names. This idea works only if all students have different last names. If we have two students with the last name Johnson, we cannot describe the last names as a set, because a set does not allow repetitions.

3.2. What is a bag?

If we allow repetitions, we arrive at the concept of a *bag* (*multiset*). Informally, a *bag* is a collection of elements over some domain. Unlike sets, bags allow *multiple* occurrences of elements. For example $\{a, a, b\}$ is a bag but not a set.

Comment. To save space on repetitions while describing a bag over some domain D, it is sufficient to describe how many times each element of D enters this bag. E.g., a bag $\{a, a, b\}$ can be written as $\{a^{|2}, b\}$. This idea helps to describe bags in set-theoretic terms [13]:

Definition 1. *Let D be a set. A bag over D is a function $A : D \to N$ from D to the set N of non-negative integers.*

Example. If $D = \{a, b, c, d\}$, then a bag $\{a, a, b\}$ is represented by the following function: $A(a) = 2$, $A(b) = 1$, $A(c) = A(d) = 0$. This sure is an awkward representation.

Comment. If this function A takes only values 0 and 1, then it is a characteristic function of some set (namely, the set of all elements d for which $A(d) = 1$). So, sets are examples of bags, and bags are natural generalizations of sets. But sets are a basic type, while bags (their closest relatives) are treated in set languages as second-class citizens.

3.3. Solution: add bags to the language

When we write a specification, we do not want to reformulate bags into set language. It would be nicer to add bags to the list of basic types. "Add" may be the wrong word: if we already have bag as a datatype, then we do not need to have sets as another datatype, because sets are a particular case of bags.

So, the solution is to use bags, and not sets, as a basic datatype.

3.4. Bags are used in computer science:

- In many sorting problems (e.g., in databases when we sort records), we start with a list (or a set of lists) that may well have repetitions in it. Unless we specifically formulate the task of avoiding these repetitions, we can just sort them. The main sorting algorithms (see, e.g., [14]) can actually be applied to sort lists with repetitions (i.e., bags), and not only sets of data.

- Bags are used to describe Petri nets ([5, 15, 16]): namely, a state of a Petri net at any given moment of time is described by specifying how many tokens there are in each location. So, a state is a bag of locations.

- Bags are not only a good way to describe algorithms, but also a good way to describe specifications (see, e.g., [12]), because an unsorted collection of elements with possible repetitions is a frequent example of input.

3.5. More complicated data types can be easily simulated in terms of bags

Let's just give two examples:

Example 1. An *array* (e.g., [1.3, 1.4, 3.5]) can be represented as bag whose components are pairs (value, index) (in this example: $\{(1.3, 1), (1.4, 2), (3.5, 2)\}$.

Example 2. A *2-dimensional array* can be represented as a bag whose components are pairs (bag-of-values-of-a-row, row number).

For example, a matrix

$$\begin{pmatrix} 1.3 & 1.5 \\ 2.4 & 2.6 \end{pmatrix}$$

with rows $[1.3, 1.5]$ and $[2.4, 2.6]$ is represented as a bag $\{(B_1, 1), (B_2, 2)\}$, where $B_1 = \{(1.3, 1), (1.5, 2)\}$ and $B_2 = \{(2.4, 1), (2.6, 2)\}$.

3.6. Bags are natural in representing algorithms

Let's give two examples:

Example 1.

- *Problem:* Assume that we have a bag of numbers, and we want to *find the biggest* of them.
- *Algorithm:* The algorithm is simple: a processor picks a pair, compares the elements of this pair, and deletes one that is smaller (or equal). Then it picks another pair, etc. At the end, we are left with one element only: the desired biggest number.

Example 2.

- *Problem: to compute the sum of all the elements* of a given *bag* of numbers.
- *Algorithm:* We let a processor pick a pair, add the two numbers from this pair, and replace the two added numbers by their sum. At the end, when a single element remains, this element is exactly equal to the sum of all the elements from the original bag.

Let us give a numerical example. Suppose that we start with a bag $\{3, 3, 5, 4\}$. Then, the following is a possible trace of this algorithm:

- add elements 3 and 5 of the original bag; as a result, we get an updated bag $\{3, 8, 4\}$;
- add elements 3 and 8 of the current bag; as a result, we get a bag $\{11, 4\}$;
- add 11 and 4 of the current bag; as a result, we get a bag $\{15\}$ that consists of a single element 15. Therefore, this element is equal to the desired sum.

3.7. Bag languages

Names and references. Since more complicated data structures and algorithms can be naturally expressed in bag terms, bags have been proposed as a basic data type for a new generation of high-level programming languages:

- GAMMA [1–3];
- BagL (Bag Language) [6–11];
- Nesl (Nested Data-Parallel Language) [4].

What is a bag program? Main idea. Every "local" algorithm like the ones that we were talking about consists of the rules of the following type: if in the bag, there are elements $a, \ldots, b$, then we replace them with elements $p, \ldots, q$. For example, the algorithm that computes the maximum replaces a pair a, b with a single element $\max(a, b)$. The algorithm that computes the sum replaces a, b with $a + b$. Each of these replacements is of the type $\{a, \ldots, b\} \to \{p, \ldots, q\}$, where $\to$ stands for "replace".

Both lists $a, \ldots, b$ and $p, \ldots, q$ can have repetitions, and therefore, they can be viewed as bags (subbags of the big bag that we are processing). An *algorithm* can be formulated now as a sequence of such rules.

Example. If we know that the bag consists only of 0's and 1's, then an algorithm for finding the maximum can be reduced to only 4 rules: $\{0,0\} \to \{0\}$, $\{0,1\} \to \{1\}$, $\{1,0\} \to \{1\}$, and $\{1,1\} \to \{1\}$.

Simplest bag programs: a formal definition. The simplest bag programs can be described as follows:

Definition 2. *Assume that a finite set D is given. This set will be called a domain. By a rule R, we mean an expression of the type $I \to O$, where I and O are bags over D. By a simple bag program P, we mean a finite set of rules $R_1, \ldots, R_r$ (i.e., expressions $I_1 \to O_1, \ldots, I_r \to O_r$).*

Example: To compute the sum of all elements of a bag, we need the following rules: $\{1,1\} \to \{2\}$, $\{1,2\} \to \{3\}$, $\{1,3\} \to \{4\}$, $\{2,2\} \to \{4\}$, $\ldots$, $\{3,5\} \to \{8\}$, $\ldots$, $\{3,8\} \to \{11\}$, $\ldots$

Remark. This is a very simplified version of the bag language program; programs can be much more complicated. In this paper, however, we will consider only simple bag programs. So, without risking confusion, we will call them simply programs.

Definition 3. *Assume that a bag B is given. We say that a rule $I \to O$ is applicable to a bag B, if I is a subbag of B (i.e., if I can be obtained from B by deleting some of B's elements). By a result of applying a rule $I \to O$ to a bag B, we mean a bag $(B - I) \cup O$ (i.e., we delete all elements of I from B, and replace them with O).*

Example. A rule $\{3,5\} \to \{8\}$ is applicable to the bag $\{3,3,4,5\}$, and the result of this application is a new bag $\{3,4,8\}$.

Definition 4. *By a trace of applying a program P to a bag B, we mean a sequence (finite or infinite) of pairs $(R^{(i)}, B^{(i)})$, $i = 0, 1, \ldots, N$, $N \leq \infty$, where:*

- *$B^{(0)} = B$.*
- *For every i, $R^{(i)} \in P$ is a rule from the program P, and $B^{(i+1)}$ is the result of applying rule $R^{(i)}$ to a bag $B^{(i)}$. And*
- • *If N is finite ($N < \infty$), then no rule is applicable to $B^{(N)}$ (i.e., we stop only if none of the rules is applicable).*

When $N < \infty$, the bag $B^{(N)}$ is called a result (or, a possible result) of applying P to B.

Comments.

1. According to our definitions, the total number of rules in a program is always finite. However, one and the same rule can be applied many times: for example, when we find the biggest element in a bag that contains all 1's, then every time we apply a rule, it is the rule $\{1,1\} \to \{1\}$. In this particular example, the total "length" N of the trace is still finite. But in general, one can imagine cases when, e.g., one and the same rule can be applied infinitely many times (e.g., if a program consisting of only one rule $\{1\} \to \{1,1\}$ is applied to a bag $\{1\}$, it will add 1's forever). In such cases, the trace can be infinite.
2. In general, the result can depend on the trace.

4. Concurrency naturally appears in bag-processing algorithms

For the detailed description, see, e.g., [3]. We will illustrate this idea on the above two examples:

Example 1. Assume that we have a bag of numbers, and we want to *find the biggest* of them. If we have several processors that can access the bag, then the algorithm is simple: each processor picks a pair, compares the elements of this pair, and deletes one that is smaller (or equal). Then they pick another pair, etc. At the end, we are left with one element only: the desired biggest number.

Example 2. Similarly, if we want *to compute the sum of a* given *bag* of numbers, we let each processor pick a pair, add the two numbers from this pair, and replace the two added numbers by their sum. At the end, when a single element remains, this element is exactly equal to the sum of all the elements from the original bag.

Let us give a numerical example: Suppose that we start with a bag $\{3, 3, 5, 4\}$, and that we have several processors that can run in parallel. Then, the following is a possible trace of this algorithm:

- First processor picks elements 3 and 5 from the original bag and substitutes the sum 8 of these two elements instead of them. Simultaneously, the second processor picks the remaining two elements 3 and 4, and instead of them, substitutes their sum 7. As a result, we get an updated bag $\{7, 8\}$;
- One of the processors adds elements 7 and 8 of the current bag. as a result, we get a bag $\{15\}$ that consists of a single element 15. Therefore, this element is equal to the desired sum.

Comments.

1. The general feature of such parallelizations is that each processor performs some *local* operations that involve only a few elements of the original bag. The reason for preferring local operations is that the initial bag (e.g., a bag of records) may be physically located in different places, and it will be very complicated and time-consuming to let each processor retrieve the records from all these places. Besides, if we restrict each processor's access to a few elements from the bag, we thus diminish the number of hard-to-deal cases when several processors try to process the same element.
2. Several other algorithms can be similarly parallelized [1].
3. Let us now give an interval-like example:

Example 3.

- *Given:* $X = \{1, 3, 5\}$, $Y = \{2, 7\}$.
- *To compute:* $X + Y$.

- *Algorithm:*
  - First, we form the bag of all pairs; in our case,

$$X \times Y = \{(1,2),(3,2),(5,2),(1,7),(3,7),(5,7)\}.$$

  - Then, if a processor sees a pair, it can replace it by its sum. If we have 6 processors that can work in parallel, then each processor can grab and process its own pair. As a result, we get the desired bag $X + Y = \{3,5,7,8,10,12\}$.

5. Bag languages can be also applied to normal interval computations

In the computer, an interval $[a,b]$ is a pair of real numbers. In terms of bags, it is a bag $\{a,b\}$ consisting of these two numbers. All four arithmetic operations $*$ with intervals can be reformulated as follows: if an interval $X = [x^-, x^+]$ is represented as a bag $b(X) = \{x^-, x^+\}$, and an interval $Y = [y^-, y^+]$ is represented as a bag $b(Y) = \{y^-, y^+\}$, then a bag $b(X * Y)$ that represents an interval $X * Y$ can be obtained if we do the following:

- form a bag of pairs $b(X) \times b(Y)$ with elements $\{(x^-, y^-), (x^-, y^+), (x^+, y^-), (x^+, y^+)\}$;
- substitute each pair (x, y) from the bag $b(X) \times b(Y)$ with the result $x * y$ of applying the operation $*$ to x and y; as a result, we get a bag consisting of four elements $\{x^- * y^-, x^- * y^+, x^+ * y^-, x^+ * y^+\}$;
- in this resulting bag, leave only the biggest and the smallest elements.

Comment. Actually, this procedure leads to a correct interval result:

- for addition $(+)$, subtraction $(-)$, and multiplication $(\times)$: in all cases;
- for division $(/)$: in all cases in which the result is an interval (i.e., when $0 \notin Y$).

Acknowledgments. This work was sponsored by the Air Force Office of Scientific Research (AFSC), under contracts F49620–89–C–0074 and F49620–93–1–0152, by NSF Grant No. CDA–9015006, and NASA Research Grant No. NAG 2–670 Supplement No. 2.

References

[1] Banâtre, J.-P., Courant, A., and Métayer, D. Le *A parallel machine for multiset transformation and its programming style.* Future Generation Computer Systems **4** (1988), pp. 133–144.

[2] Banâtre, J.-P. and Métayer, D. Le *The GAMMA model and its discipline of programming.* Sci. Comput. Program. **15** (1990), pp. 55–77.

[3] Banâtre, J.-P. and Métayer, D. Le *Programming by multiset transformation.* Communications of the ACM **36** (1) (1993), pp. 98–111.

[4] **Blelloch, G. E. and Sabot, G. W.** *Compiling collection-oriented languages onto massively parallel computers.* Journal of Parallel and Distributed Computing **8** (2) (1990), pp. 119–134.

[5] Cerf, V., Fernandez, E., Gostelow, K., and Volansky, S. *Formal control flow properties as a model of computation.* Report ENG–7178, Computer Science Department, University of California at Los Angeles, 1971.

[6] Cooke, D. E. and Gutierrez, A. *An introduction to BagL.* In: "IEEE Fourth International Conference on Software Engineering and Knowledge Engineering", Capri, Italy, 1992, pp. 479–486.

[7] Cooke, D. E. *Arithmetic over multisets leading to a high level language.* In: "Proceedings of the Computers in Engineering Symposium", Houston, TX, 1993, pp. 31–36.

[8] Cooke, D. E. *Possible effects of the next generation programming language on the software process model.* International Journal of Software Engineering and Knowledge Engineering **3** (3) (1993), pp. 383–399.

[9] Cooke, D. E. *A high level computer language based upon ordered multisets.* In: "Proceedings of the IEEE Fifth International Conference on Software Engineering and Knowledge Engineering", San Francisco, 1993, pp. 117–124.

[10] Cooke, D. E. *An executable high level language based on multisets.* Submitted to IEEE Transactions on Software Engineering, to appear.

[11] Cooke, D. E., Duran, R., Gates, A., and Kreinovich, V. *Bag languages, concurrency, Horn logic programs, and linear logic.* In: "Proceedings of the Sixth International Conference on Software Engineering and Knowlegde Engineering SEKE'94, June 21–23 1994, Jurmala, Latvia", IEEE Computer Society and Knowledge Systems Institute, Skokie, IL, 1994, pp. 289–297.

[12] Dromey, G. *Program derivation. The development of programs from specifications.* Addison-Wesley, Sydney, 1989.

[13] Gries, D. and Schneider, F. B. *A logical approach to discrete math.* Springer-Verlag, N.Y., 1993.

[14] Knuth, D. *The art of computer programming. Seminumerical algorithms.* Addison-Wesley, Reading, MA, 1969.

[15] Petersen, J. L. *Computation sequence sets.* Journal of Computer and System Sciences **13** (1) (1976), pp. 1–24.

[16] Peterson, J. L. *Petri net theory and the modeling of systems.* Prentice-Hall, 1981.

[17] Schwartz, J. *Programming with sets: an introduction to SETL.* Springer-Verlag, N.Y., 1986.

Received: November 20, 1993
Revised version: June 2, 1994

Department of Computer Science
University of Texas El Paso
El Paso, TX 79968
USA
E-mail: dcooke@cs.utep.edu