RGA Users Manual

Version 2.3

by

E. Timothy Morgan

ABSTRACT

RGA is an interpreter for a special language designed for the analysis of reachability graphs, or control flow graphs, generated from Petri nets. Although in some cases the reachability graph can become too large to be tractable, or can even be infinite, many interesting problems exist whose reachability graphs are of reasonable size. In RGA, the user has access to the names of the places in the net, and to the states of the reachability graph. The structure of the graph is also available through functions which return the sets of successor or predecessor states of a state and the transition-firings connecting the states. The RGA language allows dynamic typing of identifiers, recursion, and function and operator overloading. Rather than providing a number of predefined analysis functions, RGA provides primitive functions which allow the user to conduct complex analyses with little programming effort. RGA is part of a suite of tools, called P-NUT, intended to facilitate the analysis of concurrent systems described by Petri nets.

Technical Report #87-04

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

January 13, 1987

Contents

RGA Users Manual
Version 2.3

1. Introduction

RGA is an interpreter for a special language designed for the analysis of reachability graphs, or control flow graphs, generated from Petri nets [PETE77]. Although in some cases the reachability graph can become too large to be tractable, or can even be infinite, many interesting problems exist whose reachability graphs are of reasonable size. In RGA, the user has access to the names of the places in the net, and to the states of the reachability graph. The structure of the graph is also available through functions which return the sets of successor or predecessor states of a state and the arcs connecting the states. The RGA language allows dynamic typing of identifiers, recursion, and function and operator overloading. Rather than providing a number of predefined analysis functions, RGA provides primitive functions which allow the user to conduct complex analyses with little programming effort. RGA is part of a suite of tools, called "P-NUT" (Petri Net UTilities), developed by the Distributed Systems group at UC Irvine. The P-NUT tools are intended to facilitate the analysis of concurrent systems described by Petri nets.

In RGA, the user merely types an expression, and the interpreter evaluates it and prints the resulting value. For example, using the function **nsucc** which returns the number of successor states of a state, and the set of all states S, the user can write

```
forall s in S [nsucc(s) > 0]
```

This expression will return **true** if for each state in the set S, the number of its successors is greater than zero. Thus this expression is a test for deadlock-freeness of the Petri net [AGER79].

Another test might be to determine if the net is conservative, that is, that tokens are never gained or lost [AGER79]. The function `tokens(`*s*`)` returns the sum of the tokens on all places in a state *s*. The first state in the graph is written #0, so the expression for net conservation might be

```
forall s in S [tokens(#0) = tokens(s)]
```

The following sections describe the properties of the interpreter for the language, the data types and expressions which exist in the language, and how the user may define functions using the primitive functions provided by the interpreter. Then some examples are given to show how the system may be used to answer more complex questions than can be answered using the primitive functions. Finally, some implementation issues are discussed and some conclusions are drawn.

2. Execution Environment

RGA is an interpreter, and thus its operation is similar to that of most LISP interpreters. Any expression which the user types is immediately evaluated, and that value is printed on the standard output. The expression is then thrown away, and the user is prompted again for another command. In addition to typing expressions, the user may define expressions to be evaluated later as functions. Expressions and function definitions may be read from a file as well as from the standard input.

Unlike LISP, RGA has a number of distinct data types which it uses. But there is no explicit way to declare variables. In fact, all variables in the RGA language are dynamically typed when they are assigned values: an identifier or expression always represents a <*value, type*> pair. The user never explicitly deals with the *type* component, however. During execution, an identifier may have more than one value (and therefore, type) associated with it simultaneously. These values are stored on an execution stack, and only the most recently bound value may be accessed at any time.

Because identifiers need not be declared before their use, it is very easy to define functions. However, it also means that much of the type checking which needs to be performed must be delayed until execution time, since the types and values of identifiers used in a function definition will not be known at the time the function is defined.

Three types of errors are possible using RGA. The first error type is a syntax error in an expression or command. This type of error results in the message "`Command ignored.`" The second type of error is a run-time error, such as a type conflict or a division by zero. A run-time error usually results in an appropriate message's being printed, followed by a prompt. The execution stack is *not* "cleaned up" so that variables will have the values they had at the time of the error, facilitating debugging of defined functions. If user-defined functions were being executed at the time of the error, a stack-back trace of function calls is printed. The final type of error is an internal error in the RGA interpreter, which should not happen under normal circumstances. Usually this type of error prints a message and produces a core image for debugging the interpreter.

3. Lexical Issues

RGA is case sensitive. All command keywords and predefined function names are written in lower case. All identifiers which the user defines may be written in lower, upper, or mixed case. The user may not redefine a reserved language keyword, but predefined identifiers may be redefined, although that is not recommended. In addition to the five predefined identifiers `S`, `P`, `T`, `C`, and `A` (described later), the reachability graph which is loaded during initialization will typically define a number of identifiers to be places and sets of places and transitions; these identifiers must follow the requirements for identifiers described below.

An identifier is represented as an upper or lower case alphabetic letter, followed by zero or more letters, digits, single-quote characters, periods, and underscores. A number is represented as an optional minus sign followed by one or more digits; floating point as well as integer values may be represented.

A command to RGA is normally terminated by a newline character. Receiving this character will cause RGA to perform the indicated function. For very long expressions or function definitions, a line may be terminated with a backslash (\) followed by a newline. This combination of two characters is treated as a single space character, so its only effect is to delimit other tokens. Multiple space and tab characters and comments are treated as a single space. Comments may be inserted using the conventions of the C and PL/I languages: /* *comment text* */.

4. Expression Types and Execution Semantics

This section describes the syntax and semantics of the expressions available in RGA, and it describes the built-in primitive functions which are available. It is divided into subsections which describe each of the different data types which the language supports and the functions which return those types. The commands which may be used to define new functions are described in Section 4, and a formal BNF description of the language is given in Appendix A. All expressions in the RGA language evaluate to a value whose type is either a state, an integer, a boolean, a transition, an arc (A), a floating point number, a character string, a set, or a sequence. Identifiers can be assigned values of any of these types, and they will automatically take on the appropriate type. Strings are denoted by enclosing the characters of the string in double quotation marks. Floating point constants are indicated by the presense of a decimal point. The arithmetic operations which may be applied to integer values may also be applied to floating point values, but mixed argument types are not allowed. The built-in functions `int` and `float` may be used to convert between integer and floating point values.

Normally, evaluating a place is interpreted to mean the integer number of tokens on that place within the context of a particular state. If no state is specified explicitly, then the place value will be returned instead.

Calls to functions, both those which are predefined, and to those defined by the user, have the same syntax:

id(*list-of-expressions*)

where a *list-of-expressions* is a single expression, or multiple expressions separated with commas. If the function takes no arguments then the parentheses are omitted. When invoking a function, the expressions are evaluated from right to left in the current context, and then all the values are bound to the formal parameters, from left to right. Thus all parameters are passed by value, so they cannot be changed in any way by the called function unless it accesses them globally. Variable bindings are dynamically scoped.

When evaluating a place identifier, as mentioned above, its value is the integer number of tokens on that place in a certain state of the control flow graph. To specify the state to use explicitly, the place identifier should be followed by a state-valued expression in parentheses.

There are four special expression operators whose type depends on their arguments, and which operate on expressions of all types. (1) The `print` function takes an arbitrary expression in parentheses, and it returns the value of that argument. It has a side-effect of printing the value returned on the standard output. (2) The infix assignment operator "`:=`" assigns the identifier on its left the <*value, type*> pair which results from evaluating the expression on its right. This operator is right-associative. Like `print`, the assignment operation also returns the value which has been evaluated. (3) The semicolon infix operator "`;`" takes two expressions of arbitrary type. It evaluates the left expression and discards its value, if any, and then it returns the value of the right hand expression. The semicolon operator is left-recursive in its evaluation. (4) Finally, the `if` expression allows for conditional execution of expressions. There are two forms of the `if` command:

> `if` *boolean-expression* `then` *expression* `fi`
>
> `if` *boolean-expression* `then` *expression* `else` *expression* `fi`

The type and value returned by the `if` expression depends on what expression, if any, is executed. It is unique, however, in that it may not return a value at all if the *boolean-expression* in the first form evaluates to `false`. The only time that this form of the `if` expression can be used is as the left argument to a semicolon operator, which would discard any value returned. The `else`-less form is not allowed in any other situation, when a value is required.

4.1. Integer Expressions

Integer expressions follow the conventions of most modern programming languages. An integer value may be an integer constant, an identifier whose value is an integer, a place (when it is evaluated as the number of tokens on that place, as explained above), or an integer-valued function. A number of arithmetic functions are written in conventional infix notation: addition (+), subtraction (-), multiplication (*), division (/), modulo (%), and exponentiation (^). Unary negation is recognized. Parentheses can be used to control the evaluation of an expression; conventional precedence and left-to-right evaluation order otherwise hold.

The following are the predefined integer-valued primitive functions. The argument types (states and sets) are described in later sections.

`int(`*float*`)`	The expression *float* is evaluated, which should result in a floating point value. This value is then rounded to an integer value, which is returned as the value of the `int` function. There is a corresponding `float` function which converts values the other way.
`tokens(`*state*`)`	The total number of tokens on all places in a specified state. The state is given as an argument to the function, as `tokens(#0)`. An alternate way of writing this function is to put the state within vertical bars, as an absolute value. For example, `|#1|`.
`marked(`*state*`)`	Returns the number of places in the argument state which have at least one token on them. If `marked(`*s*`)=tokens(`*s*`)` then the state *s* is a safe state [AGER79].

`nsucc(`*state*`)`	Returns the number of successor states of the argument state. If there are two or more arcs which lead to the same state, then `nsucc` will actually return the cardinality of the `aout` set rather than that of the `succ` set.
`npred(`*state*`)`	Returns the cardinality of the `ain` set, as `nsucc` returns the cardinality of the `aout` set.
`card(`*s*`)`	Returns the number of elements of a set or sequence *s*, which is the single argument.
`capacity(`*place*`)`	Returns the capacity of a place as defined in the Petri net. Returns 0 when this information was not specified.

4.2. Floating Point Expressions

The arithmetic and relational operators described above may also be applied to floating point values. A floating point constant is differentiated from an integer constant by the presense of a decimal point. Standard exponential notation may also be used. Floating point and integer values may not be mixed in use with the arithmetic or relational operators. The built-in functions which return floating point values are:

`float(`*int*`)`	Converts the integer expression *int* to a floating point value.
`enable_time(`*trans*`)`	Returns the enabling time for a transition *trans*. Zero is returned for untimed Petri nets.
`firing_time(`*trans*`)`	Returns the firing time for a transition *trans*. Zero is returned for untimed Petri nets.
`prob(`*trans*`)`	Returns the probability of a transition *trans* as a floating point value. Returns 0.0 when a probability has not been assigned in the Petri net.
`atime(`*a*`)`	Returns the time associated with an arc *a* in a timed reachability graph as a floating point value. Returns 0.0 with untimed graphs.
`aprob(`*a*`)`	Returns the probability associated with an arc *a* in a timed reachability graph as a floating point value. Returns 0.0 with untimed graphs.

4.3. Boolean Expressions

As with other expressions, boolean expressions are built up from constants, infix operators, and predefined and user-defined function calls. The boolean constants are the reserved words `true` and `false`.

The infix boolean operators are the conventional arithmetic comparison tests: `<`, `<=`, `>`, `>=`, `=`, and `!=`; equal may also be written `==`, while not equal may also be written as `<>`. The equal and not equal tests may be applied to any data types (places, states, sets, sequences, and booleans, as well as integer-valued expressions), while the other operators are restricted to integer, floating point, and string expressions. Of course, the `=` operation applied to boolean expressions is a logical equivalence test. Other infix boolean operators which apply to boolean expressions are `implies`, `iff`, `and`, and `or`. For convenience, these operators may also be written as `=>`, `<=>`, `&` and |, respectively. Both the `and` and `or` operators are "short-circuit" operators which evaluate the lefthand operand, and then only evaluate the righthand operand if necessary. Their precedence, from lowest to highest, is the logical relational operators `implies` and `iff`, which have neither left nor right associativity, `and` and `or`, which are left associative, and the arithmetic relational operators, which also have no associativity. The prefix unary operator `not` may be used to negate a logical expression. It has the precedence of arithmetic relational operators, so it is higher than the other logical operators.

As a special case, places may be used as boolean values if they contain at most one token. This is for convenience when working with safe nets. When a boolean expression is expected, and a place name is found instead, then the number of tokens on that place is evaluated, and `false` is returned if it is zero, and `true` if it is one.

Two of the language's most important operators are `forall` and `exists`, the universal and existential quantifiers. Their syntax is the same, so only that of `forall` will be given:

`forall` *id* `in` *set-expression* [*boolean-expression*]

This expression is evaluated as follows. First, the current value of *id* is pushed on the execution stack, to be popped off when the `forall` expression is finished being evaluating. The global current state (the value of `C`) is also pushed and popped at the same time if the set is a set of states. Next, the *set-expression* is evaluated once and only once. The *id* is then looped through the elements of the set one at a time. If the *id* is a state, then the current state is set to be that state also. For each value of the *id,* the *boolean-expression* is evaluated. If for all values, the expression evaluates to `true`, then the whole expression returns that value. But if the expression ever evaluates to `false`, then execution of the loop is halted immediately and the `forall` expression returns `false`.

The `exists` expression is similar to `forall`, but with the logical tests reversed. It continues to evaluate the boolean expression until it exhausts all the elements of the set or until the expression evaluates to `true`. If the set is exhausted, then `exists` returns `false`, and otherwise, `true`.

There are five primitive functions which return a boolean value. The first, `in`, determines whether a value is an element of a set or sequence. The remaining four are *branching time temporal logic* functions. Each of these four functions takes three arguments: a state q and two expressions f and g which are expected to return boolean values. These functions and their descriptions, were inspired by functions with the same names described in [FERN85].

`in(`*item*, *s*`)`	The `in` function takes two arguments, an *item* of any type, and a set or sequence of items *s*. It returns `true` if the item is an element of the indicated set or sequence, and `false` otherwise.
`all(`q, f, g`)`	Returns `true` if for all states subsequent to state q, f is `true` as long as g is `true`.
`inev(`q, f, g`)`	Returns `true` if for all successor states of state q, f is `true` sometime before g becomes `false`.
`pot(`q, f, g`)`	Returns `true` if for some sequence of successor states of state q, f becomes `true` before g becomes `false`.
`some(`q, f, g`)`	Returns `true` if on some sequence of successor states of state q, f is `true` until g becomes `false`.

Quite often, the third argument to the temporal logic functions will be the boolean constant `true`. For example, the expression

`forall s in allsucc(`*state*`) [`*expr*`]`

can be more quickly computed (especially when its result is `false`) using the expression

`all(`*state*, *expr*, `true)`

because the latter function performs a depth-first search over the successor set of *state*, stopping as soon as it discovers a state in which *expr* is `false`.

As each of these functions is evaluated, part or all of the reachability graph will be traversed. The value of `C` will be set to each state visited in turn. This value may then be referenced in the boolean expressions f and g.

4.4. State, Place, and Transition Expressions

State constants may be written as a pound sign (`#`) followed by an integer. The first state in a complete reachability graph is `#0`. Places can only be referred to through the identifiers defined in the original Petri net from which the reachability graph is derived, and through loop control identifiers in the `forall` and `exists` expressions, and the *subset* construct described in the following subsection.

Unnamed transition constants are written as a dollar sign (`$`) followed by an integer, with the first transition written as `$0`. Those which have been given a name in the Petri net will be identified by that name instead.

A state in the reachability graph consists of a marking of the places of the net, a set of arcs to other states, and a set of arcs coming from other states. In addition, for timed nets, the state will contain remaining enabling and firing times for some of the transitions. This time information is returned by the `ret` and `rft` functions. The marking of the places in a state may be easily displayed using the `showstate` function:

`showstate(`*state*`)`	Returns the state argument, and prints its marked places as a side effect. If more than one token is on a place, the token count is shown in parentheses.

4.5. Set and Sequence Expressions

The set operations are probably the single most powerful feature of the language. Sets and sequences are composed of elements which must be of the same type. Any legal type is acceptable, including other sets and sequences; all the elements of a set or sequence must be of the same type. Although sets should be considered to be unordered, they are always maintained in ascending numerical order for convenience in reading and comparing them. Sets do not contain duplicate elements, while sequences are ordered and may contain duplicates. A single set is either a set variable, a set constant, or a set function.

Four predefined set variables exist: `S`, `P`, `T`, and `A`. The set `S` is the set of all states in the reachability graph, and `P` is similarly the set of all places in the original Petri net. `T` is the set of all transitions in the Petri net potentially-firable from the initial state. `A` is the set of all arcs in the reachability graph. In addition to these four variables, any place or transition arrays defined in the Petri net will be represented in RGA as sets of places or transitions.

Variables whose values are a set or sequence may be indexed to pick individual members of the set. The first element is numbered 0. Only variables may be indexed in this manner. The index expression, which should evaluate to an integer, is given in parentheses following the identifier.

A set constant is written as a list of expressions within curly braces {}. The list is written with the elements separated with commas. For convenience, a constant range of states may also be entered by giving the first state, "`..`", and the final state of the range. For example, the set consisting of states 1, 5 through 10, and 12 could be written

```
{#1, #5..#10, #12}
```

The list of elements may be empty, resulting in the empty set. As a special case, an empty set may be used in the context of a set of any type without a type-conflict error.

Another powerful way of specifying a set constant is the *subset* construct. It allows elements to be selected from a set using any boolean expression as the selection criterion. This construct is similar to a `forall` command, but it always loops through the entire set evaluating the *boolean-expression* for each element. Like the `forall` and `exists` statements, the *id*'s value is pushed at the beginning of the loop and restored when the subset has been constructed. If the *set-expression* is a set of states, then the "current state" will also be pushed and popped, and set to each value along with the *id*. The *set-expression* is evaluated only once. The subset construct is written

{*id* `in` *set-expression* | *boolean-expression*}

Sequence constants are written just like set constants, except that their elements maintained in the order in which they were added to the sequence, and pairs of less-than (<<) and greater-than (>>) symbols are used to indicate the beginning and ending of the sequence, instead of open and close braces.

It is recommended that set and sequence constants be used only in contexts where they will not be repeatedly evaluated, as within a loop or a recursive function, because they are relatively expensive to compute. If a constant is to be used in these situations, it should be evaluated once and the value assigned to some identifier. Then that identifier may be used in place of the constant. There are several predefined functions which return sets as their values. Addition functions which return sequences will be discussed in the following subsection on arcs.

ain(*state*)	Returns the set of arcs whose destination state is the indicated *state*.
aout(*state*)	Returns the set of arcs whose source state is the indicated *state*.
succ(*state*)	The **succ** function takes a state expression as its argument. It returns the (possibly empty) set of immediate successor states in the reachability graph of the specified state.
pred(*state*)	The **pred** function is similar to the **succ** function, but it returns the set of immediate predecessor states instead of the successor set.
allsucc(*state*)	Returns the set of all the successors of the indicated state, and recursively, all their successors.
allpred(*state*)	Returns the set of all the predecessors of the indicated state, and recursively, all their predecessors.
input_places(*trans*)	Returns the set of input places for a particular transition *trans*. Since it is a set, places which have multiple arcs to *trans* will appear only once.
output_places(*trans*)	Returns the set of output places for a particular transition *trans*.
inhibitor_places(*transl*)	Returns the set of places connected to *trans* via inhibitor arcs.
union(*s1, s2*)	The **union** function takes two sets or sequences *s1* and *s2* as its arguments. Both set/sequences must consist of elements of the same type, or at least one must have zero cardinality. This function returns the set union of the two sets, or the concatenation of the two sequences in the order given. The infix plus operator (+) may be written in place of the **union** function.
intersection(*s1, s2*)	This function is similar to the **union** function, but it returns the set intersection of its two arguments which must both be sets. The two arguments must both be sets of the same type, or at least one must be the empty set.
setdiff(*s1, s2*)	The **setdiff** command takes two arguments with the same restrictions as the **intersection** function. It returns a copy of *s1* minus any elements it has in common with *s2*. Elements of *s2* which do no appear in *s1* are ignored. The **setdiff** function may be written using the infix minus (-) operator.

`setop`(*func, set*)	The `setop` operator applies the function *func,* which must be a monadic function, to each element of the *set.* The results of the function executions are unioned into the resulting set, which is returned as the value of the `setop` function. The function *func* may return values which are either individual elements or sets of elements; it may be either a user-defined function or one of the predefined functions `succ`, `pred`, `card`, `marked`, `nsucc`, `npred`, `src`, `dest`, `trans`, `allpred`, `allsucc`, `showstate`, `conflict_set`, `atime`, `probability`, `capacity`, `enable_time`, `firing_time`, `input_places`, `inhibitor_places`, `float`, `int`, `aprob`, `ain`, `aout`, and `output_places`.
`set`(*sequence*)	The *sequence* will be converted to a set of the same type, eliminating duplicate elements.
`ret`(*s, trans*)	Returns a sequence of the remaining enabling time(s) for a transition *trans* within the context of a state *s.* For untimed reachability graphs, or when no enablings of the transition exist in the state, an empty sequence is returned.
`rft`(*s, trans*)	Returns a sequence of the remaining firing time(s) for a transition *trans* within the context of a state *s.* For untimed reachability graphs, or when no instances of firings of the transition exist in the state, an empty sequence is returned.

4.6. Arcs

Arcs connect the states in the reachability graph. In the case of a timed reachability graph, an arc will consist of a *from_state,* a *to_state*, a list of transitions beginning to fire (the *tbegin* sequence), and a list of transitions ending firing (the *tend* sequence). With an untimed net, the tbegin sequence will contain the one transition whose firing moves the Petri net from the *from_state* into the *to_state,* and the tend sequence will be empty. With timed nets, the tbegin and tend sequences will each contain zero or more transitions, possibly including duplicates.

Arc constants are written as a quadruple of the source and destination states of the firing, and the tbegin and tend sequences. The components of the quadruple are written separated by commas, between square brackets. For example, `[#0, #10, <<$10>>, <<>>]` is a firing of transition `$10` taking the net from state `#0` to state `#10`. It is a run-time error to evaluate an arc constant which does not exist in the reachability graph. Four primitive functions exist which take arcs as their single argument and return the separate components of the arc:

`src(`*a*`)`	Returns the source state of *a*.
`dest(`*a*`)`	Returns the destination state of *a*.
`tbegin(`*a*`)`	Returns the sequence of transitions which begin firing with arc *a*.
`tend(`*a*`)`	Returns the sequence of transitions which end firing with arc *a*.

5. User-Defined Functions

Defining a function is similar to assigning a value to a variable. The primary difference is that the expression is not immediately evaluated. Instead, the parse tree which represents the expression is stored as the value of the identifier, with a special type indicating that the "value" of the identifier is an unevaluated expression tree. Whenever that identifier is subsequently evaluated, the expression tree is retrieved and evaluated, with its value being returned as the value of the identifier. Functions defined in this way may be written recursively; as in pure LISP, recursion is the primary mechanism for looping and flow control.

To define a function, the "`::=`" operator is used. ***Functions may be defined only at the top command level.*** If RGA is being used interactively, then defining a function will cause the message "ok" to be printed on the terminal. The list of formal parameters for the function, if any, is enclosed in parentheses after the identifier and before the `::=` operator. As with function calls, the parentheses are omitted if there are no parameters. Local variables, if any, of the function are listed within square brackets following the formal parameters. If there are no local variables, the brackets are omitted. The expression which defines the function is given to the right of the operator. At the top command level, the special command `show` *id* will print the definition of the *id* if it is a function. The full syntax of a function definition is given in Appendix A.

Like other identifiers in the language, the arguments are given their types at the time they are bound to values, when the function is invoked. For example, assume the following trivial function definition:

```
setx(v) ::= x:=v
```

Then one may type `setx(1)` and the identifier x will be defined as having the value of the integer constant 1. The `setx` function will also return the integer value 1, since it expands to an assignment expression which has that value. Subsequently typing the command

```
setx({s in S | nsucc(s)=0})
```

causes x to be assigned to the set of all deadlocked states. The previous value of x is thrown away. Note that the formal parameter v takes on values of different types dynamically. The existing value of v, if any, will still be valid when `setx` has returned. Incidently, it is the current dynamically scoped "global" value of x which is assigned in the above examples.

6. Some Examples

In the introduction, some expressions for net deadlock-freeness and net conservation are given. In this section, some more complex examples are given to illustrate the full power of the system.

If the Petri net is *safe,* then each place will have at most one token on it (i.e., each place is 1-bounded [PETE77]). One might test for this condition with the expression

```
forall s in S [forall p in P [p <= 1]]
```

Note that the value of `p` in `p <= 1` is `p(s)`. There is a faster way to test for net safety, however:

```
forall s in S [marked(s) = tokens(s)]
```

This function works because the `marked` function returns the number of places which have at least one token on them for the state s. If any of these places has more than one token on it, then `tokens(s)` will be greater than `marked(s)`. It is faster than the first expression because it avoids the doubly-nested loop and makes more use of primitive functions.

In an untimed reachability graph, it is convenient to access the one transition whose firing is represented by an arc in the graph. Such a function could be defined as:

```
trans(a)[tseq] ::= tseq := tbegin(a); tseq(0)
```

Suppose one wishes to know the maximum value of the `marked` function over all the states in the graph. This could be obtained with the function shown in

```
/*
 * Findmax returns the maximum value attained by marked() over all states.
 */
findmax[max] ::=\
    max:=0; \
    forall s in S [if marked(s) > max then max := marked(s) fi; true]; \
    max
```

Figure 1

Function to find the maximum value of marked

```
/* Return a sequence of transitions which are enabled in state s */
enabled_in(s)[result,t] ::=\
    result := <<>>;\
    forall t in setop(tbegin, aout(s)) [result:=result+t; true]; \
    result

/* Return a sequence of transitions which are firing in state s */
firing_in(s)[result,t] ::= \
    result := <<>>; \
    forall t in T [forall x in rft(s,t) [result:=result+<<t>>; true]]; \
    result
```

Figure 2

Return the set of enabled transitions in state *s*

Figure 1. Notice the use of the semicolon operator to make the expression in the `forall` statement return a `true` value, thus guaranteeing that each state in `S` will be tested. The entire expression returns the maximum value found, `max`, which is a local variable of the function.

The functions shown in Figure 2 return lists of transitions which are enabled in a particular state, or which are currently firing in some state. The lists are returned as sequences, since there can be more than one instance of a given transition in either catagory. The `enabled_in` function generates a set of all the transitions which begin firing on any output arc of the specified state *s*. It then appends all the sequences in this set into one sequence (`result`) and returns that value.

The `firing_in` function finds all the transitions which have remaining firing times in state *s*. For each remaining firing time (even if it is zero), the transition is added to the `result` list. When all the transitions have thus been tested, the `result` sequence is returned as the value of the function.

Now suppose that we wish to define a boolean function which returns `true` if a particular state can be reached from another state in the graph (see Figure 3).

```
/*
 * Breadth-first search version of reachable
 */
reachable(s, final)[morestates, tried] ::= \
    s = final | \
    (tried := emptyset := {}; \
    try({s}))

/*
 * For each s in nextset, test if final is in succ(s).
 * If not, iterate on all those successors.
 */
try(nextset) ::= \
    morestates := emptyset; \
    exists s in nextset [in(final, succ(s)) | \
                         card(morestates := union(morestates,succ(s))) < 0] | \
    card(morestates:=setdiff(morestates, tried:=union(tried, nextset))) > 0 & \
        try(morestates)
```

Figure 3

Recursive Breadth-First Reachability Function

The definition given here is breadth first: it always has a set of states that it knows have already been checked, and one whose successors have not yet been checked. The function is actually divided into two parts, `reachable(`*s*`,` *final*`)` and `try(`*nextset*`)`. The `reachable` function is the top-level definition. It checks if the starting state, *s*, is the same as the desired state, *final.* If not, it initializes the set constant *emptyset,* which is used in the `try` function only for speed and clarity, and calls `try`.

This pair of functions takes advantage of the the dynamic binding of RGA. The `reachable` function has two local variables *morestates* and *tried.* They are both actually locals of the pair of functions, since they are shared by both. If they were not locals of `reachable`, then any global value with the same name would be lost by executing the `reachable` function. *Morestates* contains the next set of states to be tested by the `try` function; *tried* is the set of states whose successors have already been tested.

The `try` function takes one argument: the set of states which have recently been tested against *final* (*nextset*) whose successors now need to be checked. For each element of *nextset,* `try` compares its successors to *final.* This test can be made quickly since it uses only the built-in functions `exists`, `in`, and `succ`. If a match is found, `try` returns `true`. If all the matches fail, *morestates* will have been assigned the set of all the successors just tested. Any states in *morestates* which are in the set *tried* are removed, and the cardinality of the resulting set is compared to zero. If it is zero, then `try` will return `false` since there are no more states whose successors have yet to be compared to *final.* Otherwise, `try` is invoked recursively to try the elements in *morestates,* with *tried* augmented with the set of states just tried.

The `allsucc` function could have been used to determine the same function. If the matching successor is near the end of those tested by the above function, or if there is no match, it would be significantly faster than the `reachable` function given. But if the match occurs early in the search, then the above code could be significantly faster, since it would not bother to generate further successors.

As a final example, suppose that all the states in the reachability graph have been partitioned in to two sets, `good` and `bad`. One might then wish to know the transition(s) which lead from the `good` to the `bad` states. The set of arcs between the two sets is expressed as

```
arcs := intersection(setop(aout, good), setop(ain, bad))
```

The set of "critical" transitions (those which take the net from the "good" states to the "bad" ones [RAZO80]) is then given by the expression

```
setop(trans, arcs)
```

using the function `trans` defined above.

7. Running RGA

Typically, the user will enter a Petri net representation of the system to be analyzed in a symbolic notation. This symbolic representation is translated into the reachability graph via other programs, described elsewhere.

If files are given on the command line, then each one is read in turn. RGA acts as if `/dev/tty` were always given after any other files given on the command line. As a special case, if no files are specified, RGA reads from *stdin* before reading from `/dev/tty`, to allow its input to be piped directly from another program.

RGA expects its input to be a "canonical" Petri net followed by a "canonical" reachability graph, followed by expressions and/or commands, followed by end-of-file. The formats of these "canonical" representations are described in separate documents. If RGA is reading from the terminal, then it prompts the user for each line. While reading a graph, the prompt is "*"; while reading commands, it is

">". Typically, RGA will be invoked with the name of a reachability graph on the command line. RGA will still read the expressions/commands from the user from `/dev/tty`. If the user wishes to input one or more files containing expressions, they can be listed on the command line following the reachability graph. The Petri net and its reachability graph could also be in separate files (although currently the P-NUT tools do not produce a separate reachability graph).

It is possible to have RGA read expressions and/or commands from a file after it has begun reading from `/dev/tty`. When the end of the file is reached, it resumes reading commands from the previous input source, so input files may recursively "call" other input files. The command to read from a file is the "`@`" character, followed by the name of the file to read. If the name of the file does not follow the lexical rules for an identifier in RGA, then it must be specified as a quoted string. If a command file defines global values, it is convenient to end such commands with the ; operator so that the values will not be printed as the file is interpreted. Command files may be conveniently input to RGA by specifying them on the command line following the name of the reachability graph file, as described above.

There are two commands which may only be used at the top command level; they may not be used within defined functions. First, as mentioned previously, the user may print the definition of a function by issuing the `show` command followed by the name of the function. The formal parameters and local variables, if any, will be displayed in addition to the function's definition. The other command which may be used only at the prompting level is defining a function with the `::=` command.

8. Implementation and Performance

The RGA program is written in the C programming language running on the 4.2BSD UNIX[‡] operating system, running on a VAX-11/750 computer.[†] It has been ported to a number of other systems, including Apollo, Locus, and Sun.

The RGA system implements a minimum number of primitive operations to allow all the different operations which were desired in the initial design of the system. It has been designed also to be extensible, since the user may define new functions in terms of the primitives. In particular, the ability to pass values to

‡ UNIX is a trademark of AT&T Bell Labs

† VAX is a trademark of Digital Equipment Corporation

parameters of functions, and the existence of the semicolon operator and the `if` expression, were included expressly for extensibility purposes.

But RGA was also written to be fairly efficient. Efficiency is necessary if large reachability graphs are to be handled, and the system will only really be useful if realistically-large graphs can be analyzed in a reasonable period of time. In the interests of efficiency, some non-primitive operations which could be implemented as user-defined functions have been coded as primitive routines instead. For example, as a test, the primitive function `nsucc` was defined as `ns(s) ::= card(aout(s))`. By executing the primitive and then the defined function in a loop 75,705 times, the `nsucc` function was measured to be 4.17 times faster than the user-defined equivalent. More performance measurements are given in Appendix B.

9. Conclusions and Future Work

There is no way currently to pass a function as an argument, or to pass identifiers by-reference. This capability would be useful in defining certain general functions; the **`setop`** primitive is a special case example of its use. Functions without arguments would have to be called with an empty argument list, rather than with no argument list, as is presently the case.

Perhaps future versions of RGA will overcome these problems if they prove to be serious. Other commonly-used functions may become primitives as they are identified. New primitives to make sequences more powerful also need to be identified and implemented.

Appendix A
BNF Grammer for RGA Language

```
<formals> ::  ( <list_of_exprs> )
        |

<locals> ::   [ <list_of_exprs> ]
        |

<definition> ::  <ident> <formals> <locals> ::= <expr>

<list_of_exprs> :: <expr>
        |          <list_of_exprs> , <expr>

<transition> :: $ <number>

seqstart ::       <<

seqstop  ::       >>

addop ::          +
        |         -

mulop ::          *
        |         /
        |         %

lrelop ::         iff
        |         implies

relop ::          =
        |         >
        |         >=
        |         <
        |         <=
        |         <>
        |         !=

<sequence> ::     <seqstart> <list_of_elems> <seqstop>
        |         <seqstart> <seqstop>
        |         ret ( <state> , <state> )
        |       rft ( <state> , <state> )

<set_or_seq> ::  <set>
        |       <sequence>

<expr> ::       <sequence>
        |       set ( <expr> )
        |       atime ( <expr> )
        |       aprob ( <expr> )
        |       probability ( <expr> )
        |       capacity ( <expr> )
        |       float ( <expr> )
        |       int ( <expr> )
        |       enable_time ( <expr> )
        |       firing_time ( <expr> )
        |       input_places ( <expr> )
        |       output_places ( <expr> )
        |       <string_constant>
        |       <floating_constant>
        |       <integer_constant>
        |       <identifier> := <expr>
        |       showstate ( <expr> )
        |       true
        |       false
        |       [ <state> , <state> , <expr> , <expr> ]
        |       if <expr> then <expr> else <expr> fi
        |       if <expr> then <expr> fi
        |       <subset>
        |       print ( <expr> )
        |       <expr> and <expr>
        |       <expr> or <expr>
        |       not <expr>
        |       forall <identifier> in <set_or_seq> [ <expr> ]
        |       exists <identifier> in <set_or_seq> [ <expr> ]
        |       ( <expr> )
        |       in ( <expr> , <set_or_seq> )
        |       <expr> <relop> <expr>
        |       <expr> ; <expr>
        |       <expr> <lrelop> <expr>
        |       <expr> <addop> <expr>
        |       <expr> <mulop> <expr>
        |       <addop> <expr>
        |       <identifier> ( <list_of_exprs> )
        |       tokens ( <state> )
        |       | <state> |
              |       card ( <set_or_seq> )
              |       marked ( <state> )
              |       <expr> ^ <expr>
              |       nsucc ( <state> )
              |       npred ( <state> )
              |       src ( <expr> )
              |       dest ( <expr> )
              |       tbegin ( <expr> )
              | tend ( <expr> )
              |       # <number>
              |       <identifier>

<state> ::            <identifier>
              |       <identifier> ( <list_of_exprs> )
              |       print ( <state> )
              |       # <number>
              |       <identifier> Becomes <state>

<state_range> :: # <number> .. # <number>

<list_of_elems> :: <expr>
              |       <state_range>
              |       <list_of_elems> , <expr>
              |       <list_of_elems> , <state_range>

<setopfunc> ::        succ
              |       pred
              |       ain
              |       aout
              |       tokens
              |       marked
              |       nsucc
              |       npred
              |       src
              |       dest
              |       tbegin
              |       tend
              |       allsucc
              |       allpred
              |       showstate
              |       conflictset
              |       atime
              |       aprob
              |       probability
              |       capacity
```

```
                |       float
                |       int
                |       ret
                |       rft
                |       enable_time
                |       firing_time
                |       input_places
                |       output_places

<subset> ::             succ ( <state> )
                |       conflictset ( <expr> )
                |       allsucc ( <expr> )
                |       allpred ( <expr> )
                |       pred ( <state> )
                |       ain ( <state> )
                |       aout ( <state> )
                |       setop ( <identifier> , <set> )
                |       setop ( <setopfunc> , <set> )
                |       union ( <set> , <set> )
                |       union ( <seqconst> , <seqconst> )
                |       setdiff ( <set> , <set> )
                |       intersection ( <set> , <set> )
                |       { <list_of_elems> }
                |       { }
                |       { <identifier> in <set> | <expr> }

<set> ::                <identifier>
                |       <identifier> ( <list_of_exprs> )
                |       <identifier> becomes <set>
                |       print ( <set> )
                |       <subset>
```

Appendix B
Some Performance Measurements

The table below contains some time and space measurements of the current implementation of RGA. The problem measured was the dining philosophers problem for a varying number of philosophers, between two and eight. For each number of philosophers, the number of states in the reachability graph and the time to load the reachability graph were measured. The time is divided into user and system CPU time, in seconds. In addition, the size of the interpreter in kilobytes was measured after loading the graph but before executing any tests. Finally, two typical problems were executed, testing for the safety of the net, and determining the set of states which can reach state zero (`#0`) after zero or more transition firings. The `canreach` function is a user-coded version of the `allpred` primitive. On the average, it is about 37% as fast as the primitive function. For each of these tests, the execution time minus the load time is given, in CPU seconds.

Some Performance Measurements				
Dining n	States	Load Time	RGA Size (Kb)	Homing
2	8	0.5 + 0.6	91	0.1
3	26	0.8 + 0.6	103	0.2
4	80	1.7 + 0.7	137	0.3
5	242	4.4 + 1.2	227	1.4
6	728	13.5 + 3.1	530	6.1
7	2186	44.4 + 9.3	1605	34.6
8	6560	153.1 + 36.4	5526	455.6

Figure 4

Some Performance Measurements of RGA

References

[AGER79] AGERWALA, TILAK Putting Petri Nets to Work. *IEEE Computer* (December, 1979), 85–94.

[FERN85] FERNANDEZ, J. C., RICHIER, J. L., AND VOIRON, J. Verification of Protocol Specifications Using the Cesar System. *IFIP Workshop on Protocol Specification, Verification, and Testing* (June, 1985).

[RAZO80] RAZOUK, RAMI R., AND ESTRIN, G. Modeling and Verification of Communication Protocols: The X.21 Interface. *IEEE Transactions on Computers C-29*, 12 (December, 1980), 1038–1052.

[PETE77] PETERSON, J. L. Petri Nets. *Computing Surveys 9*, 3 (September, 1977), 224–252.