Computer-Aided Analysis of Concurrent Systems

by

E. Timothy Morgan and Rami R. Razouk

85-06

ABSTRACT

The introduction of concurrency into programs has added to the complexity of the software design process. This is most evident in the design of communications protocols where concurrency is inherent to the behavior of the *system*. The complexity exhibited by such software systems makes more evident the needs for computer-aided tools for automatically analyzing behavior.

The Distributed Systems project at UCI has been developing a suite of tools, based on Petri nets, which support the design and evaluation of concurrent software systems. This paper focuses attention on one of the tools: the reachability graph analyzer (RGA). This tool provides mechanisms for proving general system properties (e.g., deadlock-freeness) as well as system-specific properties. The tool is sufficiently general to allow a user to apply complex user-defined analysis algorithms to reachability graphs. The alternating-bit protocol with a bounded channel is used to demonstrate the power of the tool and to point to future extensions.

Technical Report #85-06

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

February 8, 1985

Contents

Computer-Aided Analysis of Concurrent Systems

ABSTRACT

The introduction of concurrency into programs has added to the complexity of the software design process. This is most evident in the design of communications protocols where concurrency is inherent to the behavior of the *system*. The complexity exhibited by such software systems makes more evident the needs for computer-aided tools for automatically analyzing behavior.

The Distributed Systems project at UCI has been developing a suite of tools, based on Petri nets, which support the design and evaluation of concurrent software systems. This paper focuses attention on one of the tools: the reachability graph analyzer (RGA). This tool provides mechanisms for proving general system properties (e.g., deadlock-freeness) as well as system-specific properties. The tool is sufficiently general to allow a user to apply complex user-defined analysis algorithms to reachability graphs. The alternating-bit protocol with a bounded channel is used to demonstrate the power of the tool and to point to future extensions.

Introduction

With the increased use of distributed processing in a wide range of applications, a need exists for techniques which can be used to assist in evaluating the correctness of concurrent software/hardware. A variety of novel specification and verification approaches are being investigated ranging from highly abstract and mathematical approaches (e.g., temporal logic [4, 12]) to approaches which closely mirror and restrict implementations (e.g., algorithmic specifications [15]). All the techniques in question are based on some formal model of computation and, in order to be used effectively, must be supported by automated tools.

In recent years, the Petri Net model [8] has been used extensively to verify properties of concurrent systems. This model is particularly interesting since it

supports verification of correctness [16, 2] and evaluation of performance [13, 17, 11]. It is this versatility which motivates this paper's focus on Petri Nets. A variety of techniques have been developed to allow Petri Net models to be used to prove the partial correctness of concurrent systems. The techniques can be divided into three general classes:

1. Techniques based on exhaustive exploration of the state space. In these techniques all control states reachable from the initial state are calculated. Since a reachability graph contains all the states of the system and all the transition between the states, it can be used to prove invariant properties of the model (properties which hold over all of the states) and to prove properties related to state transitions (e.g., deadlock freeness). Past work in this field has focused on proving general properties such as deadlock-freeness [9] by relying on Petri Net properties such as liveness and boundedness. System-specific properties are usually verified by manual examination of the state space [10]. The two major weaknesses of these techniques are: (a) reachability graphs can be infinite and, (b) even if reachability graphs are finite they can be very large and therefore difficult to build and even more difficult to process (by a human).

2. Techniques based on automated analysis of the control structure of the net to deduce important properties. These techniques attempt to solve the complexity problem of exhaustive state exploration by analyzing the net itself. Examples of such approaches are automated invariant analysis [6] and reduction [3]. In automated invariant analysis some of the invariant properties of the net are derived by analyzing a matrix representation of the net. In reduction, the net is manipulated and transformed into a smaller net whose state space preserves some key properties of the original state space (e.g., preserves deadlock-freeness). In this class of techniques, the advantage gained by computational efficiency is offset by a limitation on the properties which can be proven. For example, Keller [5] explains why invariants alone cannot be used to prove deadlock freeness. Also, reduction, while preserving deadlock freeness, compresses the state space in a way which makes proving invariant properties impossible.

3. Techniques based on inductive proofs involving control and data. Keller first presented the notion of inductive invariants as a way of proving more general system properties. His approach requires one proof per transition in a net, and is capable of dealing with data transformations as well as control flow. The burden of formulating an inductive invariant falls on the designer, but automated tools can be used to guide and check the proofs (no such tools exist to date). Proofs relating to sequences of states must be based on the notion of

a "homing state" and are more difficult. This approach can be characterized as requiring a great deal of human expertise to complete a proof while the two discussed above rely more on automated processing of the nets.

The work presented here falls in the first category. It is entirely based on building and analyzing reachability graphs and is therefore limited to systems whose reachability graph is finite (the largest graph built using our tools contains 6500 states). The novelty in this work centers on a language and a corresponding interpreter which allow a designer to give formal specifications of general correctness properties (e.g., deadlock freeness) as well as system-specific properties in a way which can be easily understood and which can be automatically verified. In addition to formal correctness proofs the tool can be used to "debug" the system and its model by providing the designer with convenient ways of focusing attention on portions of the complete state space. The tool in question (the Reachability Graph Analyzer, or RGA [7]) is part of a larger suite of tools (P-NUT, Petri Net UTilities) being developed at UCI by the Distributed Systems Project.

In Section 1 of the paper, Petri nets are *briefly* reviewed. The usefulness of reachability graphs and their limitations are also discussed there. Section 2 describes some of the most primitive capabilities of RGA. The dining philosophers problem is used as a reference example. Section 3 delves into the more advanced features of RGA which make it extensible and flexible. Section 4 presents a model of the alternating-bit protocol with bounded channels and demonstrates some of the flexibility of RGA in analyzing such complex reachability graphs.

1. Petri Nets and Reachability Graphs

Petri nets [8] are bipartate, directed graphs whose nodes are transitions and places. The arcs of the graph denote those places which are *inputs* to the transitions and those which are *outputs*. Associated with each place is a number of *tokens*; a *marking* is an assignment of zero or more tokens to each place in the net. A

transition is considered to be *enabled* when there is at least one token on each of its input places.

The execution of a net involves choosing an enabled transition nondeterministically and *firing* the transition. Firing involves removing a token from each input place and placing one on each of the transition's output places. The firing operation is considered to be an instantaneous and indivisible operation, and no two transitions may fire simultaneously. If the number of output places is greater than the number of input places, then the total number of tokens in the net increases. When there is more than one token on the graph, there is a possibility of concurrent execution. However, not every token represents a separate process; some may be used only for synchronization and resource control.

A marking defines a state of a net. Thus a firing of a transition, which can result in a change in the distribution of tokens, represents a change in the state of the net. A marking μ' is said to be *immediately reachable* from a marking μ if firing some enabled transition results in changing the marking from μ to μ'. The reflexive transitive closure of the immediately reachable relationship defines the *reachability set,* the set of all markings which can be reached from μ. The reachability set of the initial marking is the set of all possible states of the net.

Since each state is reached from another state by a transition-firing, one can view the set of all states as a directed graph, with the states being nodes of the graph and the transition-firings being the arcs connecting the states. This graph is known as the *reachability graph* of the net, sometimes called the *computation flow graph.*

It is possible for the reachability graph to be infinite. This is a consequence of some place in the net acquiring an infinite number of tokens. Various techniques exist for handling this case in formal analysis of Petri nets [8], but these techniques will not be discussed here, as the analyzer being described handles only finite

graphs. Typically, problems which would be analyzed with this tool, such as communications protocols, have only a finite number of states by design.

The reachability graph can be used in a number of ways to verify properties of the Petri net. These properties include boundedness, safeness, and liveness. The maximum number of tokens on any place can be determined by examining each place in each state of the graph. If this bound is 1, then the net is safe. If the number of tokens in each state is a constant, then the net is *conservative.* If every state has at least one successor state, then the net is deadlock free. A transition is *dead* in some marking if no sequence of transition firings exists which can enable it. If there exists such a sequence, however, the transition is said to be *potentially firable.* A transition is *live* if it is potentially firable in all reachable markings. By examining the arcs of the graph, it is possible to determine if a transition is live and the set of all states which can reach or can be reached from a particular state. These properties of the net (boundedness, liveness, etc.) are important because properties of the system being modeled can be inferred from them. For example, liveness implies that the system is *deadlock free.*

The next two sections of the paper present the capabilities of the analyzer. The approach taken in the implementation of this tool is based on an understanding of the need for flexible and expandable tools. Other implementations of reachability graph builders [SARA] have tended to have a set of built-in algorithms which can be used to verify some known general properties (e.g., deadlock freeness). In this tool, more general mechanisms are made available to the user instead of specific functions which solve only certain predefined problems. Arbitrarily complex algorithms can be synthesized from the simple basic capabilities of the analyzer. Should some algorithms prove particularly useful, the analyzer can then be enhanced to provide them as built-ins.

`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 safe.
`nsucc(`*state*`)`	Returns the number arcs going out of the argument *state*.
`npred(`*state*`)`	Returns the number of arcs going into the indicated *state*.
`card(`*s*`)`	Returns the number of elements of a set *s*.

Table 1

Integer-valued Primitive Functions

2. Basic Capabilities

The RGA system functions on two levels. First, it allows the user to specify propositions and predicates [14] over a universe of discourse consisting of places and transitions in a Petri net, and states and transition-firings in its reachability graph. Variables in the propositions may be bound by assignment or by universal or existential quantification. RGA is further augmented with primitives which give it the capabilities of a simple programming language.

The intended use of RGA is to prove interesting properties about the system being modeled. Since all system behavior is encoded in the state of the system, it is critical for such a tool to allow the user to express system properties as predicates on places and states. To this end, RGA provides the user with the ability to refer to three built-in sets: the set of places in the net `P`, the set of transitions `T`, and the set of reachable states `S`. Places are referred to by their names, while states are referred to by their number (`#0` is the initial state).

`in(`*item, s*`)`	The `in` function takes two arguments, an *item* of any type, and a set of items *s*. It returns `true` if the item is an element of the indicated set, and `false` otherwise.

Table 2

Boolean-valued Primitive Function

2.1. Arithmetic Expressions

Arithmetic expressions follow the conventions of most modern programming languages, with operators such as `+`, `-`, `*`, and `/`. The operands of these operators are expressions whose values are integers. Places in the net are evaluated as the number of tokens on that place in the context of some state written in parentheses after the place name. Places may be evaluated as booleans in situations where boolean values are expected. This feature is intended for safe graphs, so the place must contain at most one token. Places which are evaluated without specifying a state context are evaluated relative to a "current state" which is set by the `forall` and `exists` operators, and by the subset construct, described below. Table 1 shows the set of predefined integer-valued primitive functions.

2.2. Boolean Expressions

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

The infix boolean operators are the conventional arithmetic comparison tests, `<`, `<=`, `>`, `>=`, `=`, and `!=`. The equal and not equal tests may be applied to any data types (places, states, sets, and booleans, as well as integer-valued expressions), while the other operators are restricted to integer expressions. Some other infix boolean operators apply to boolean expressions: `implies`, `iff`, `and`, and `or`. Both the `and` and `or` operators are "short-circuit" operators which evaluate the lefthand operand first, and then evaluate the righthand operand only if necessary.

`src(`*tf*`)`	Returns the source state of *tf*.
`dest(`*tf*`)`	Returns the destination state of *tf*.
`trans(`*tf*`)`	Returns the transition involved in the firing *tf*.

Table 3

State and Transition-valued Primitive Functions

The prefix unary operator **not** may be used to negate a logical expression. There is only one primitive function which returns a boolean value, and it is shown in Table 2.

2.3. State, Place, and Transition Expressions

States of the reachability graph and transitions in the net are numbered. Particular states can be referenced using a # symbol followed by the number of the state. Transitions are similarly referenced using a dollar sign and the transition number. Places are referred to through the identifiers defined in the original Petri net.

Arcs between nodes in a reachability graph model state transitions which result from transitions firing. These arcs, or transition-firings (*TF*s), may be referred to as a triple of the source and destination states and the transition which is fired. For example, `[#0, #10, $9]` would be the arc from state #0 to state #10 firing transition $9.

Table 3 shows the primitive functions exist which return state or transition values. All three take a transition-firing as their single argument, and return the separate components of the TF.

Below are some examples of the types of expressions which can be constructed with the capabilities described up to this point:

1. `nsucc(s) > 0`
   This expression will be `true` if state `s` has one or more successors. Such an

expression can be used to detect if `s` is deadlocked or is a proper terminal state.

2. `p1_eating(s) + p2_eating(s) + p3_eating(s) + p4_eating(s) <= 4/2`
This tests if the number of philosophers eating (modeled using appropriately named places) is less than or equal to the number of philosophers divided by the number of forks needed to eat (in state `s`).

3. `in(s, S') & marked(s) != 0`
This expression is `true` if state `s` is a member of set of states `S'` and that there is at least one marked place in `s`.

4. `tokens(s) = tokens(#0)`
This tests whether a state `s` contains the same number of tokens as the initial state. If all states satisfy this predicate, the net is said to be conservative.

2.4. Set Expressions and Operations

The set operations are the single most powerful feature of the language. Sets are composed of any legal data types, including other sets; all the elements of a single set 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. A single set is either a set variable, a set constant, or a set-valued function.

A set *constant* is written as a list of expressions (which need not be constants) within curly braces {}. For example, the set consisting of states 1, 5 through 10, and 12 can be written

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

Another powerful way of specifying a set is the *subset* construct. It allows elements to be selected from a set using any boolean expression as the selection criterion. The subset construct is written

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

`tfin(`*state*`)`	Returns the set of transition-firings whose destination state is the indicated *state*.
`tfout(`*state*`)`	Returns the set of TFs 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 reflexive transitive closure of the immediate successor set of a state.
`allpred(`*state*`)`	Returns the reflexive transitive closure of the immediate predecessor set of a state.
`union(`*s1, s2*`)`	This function returns the set union of the two sets *s1* and *s2*, which must have elements of the same type, or at least one must have zero cardinality.
`intersection(`*s1, s2*`)`	This function is similar to the `union` function, but it returns the set intersection of its two arguments, which must obey the same restrictions as in the `union` function.
`setdiff(`*s1, s2*`)`	The `setdiff` command takes two arguments with the same restrictions as the `union` function. It returns the first set minus any elements it has in common with the second set. Elements of the second set which do no appear in the first set are ignored.
`setop(`*func, set*`)`	The `setop` operator applies the monadic function *func* to each element of the *set*. The set returned by the `setop` function is the union of the results of the function executions. The function *func* may return values which are either individual elements or sets of elements; it may be either a user-defined function or a built-in one.

Table 4

Set-valued Primitive Functions

This construct creates a set of all elements in a given set *set-expression*which satisfy the property specified in the *boolean-expression.* The predefined functions which return sets are shown in Table 4.

Two of the language's most important operators are `forall` and `exists`, the universal and existential quantifiers. They allow traversal through sets, evaluating a boolean expression for each element of the set. Their syntax is the same, so only that of `forall` will be given:

`forall` *id* `in` *set* `[`*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. Next, the *set* is evaluated once and only once. The *id* is then looped through the elements of the set one at a time. 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`. Some examples of the set traversal operators follow:

1. `forall s in S [nsucc(s) > 0]`
   This predicate tests whether a net is deadlock-free.

2. `forall s in S [forall p in P [p(s) <= 1]]`
   This expression is `true` if all the places in the net are safe (1-bounded) in all states.

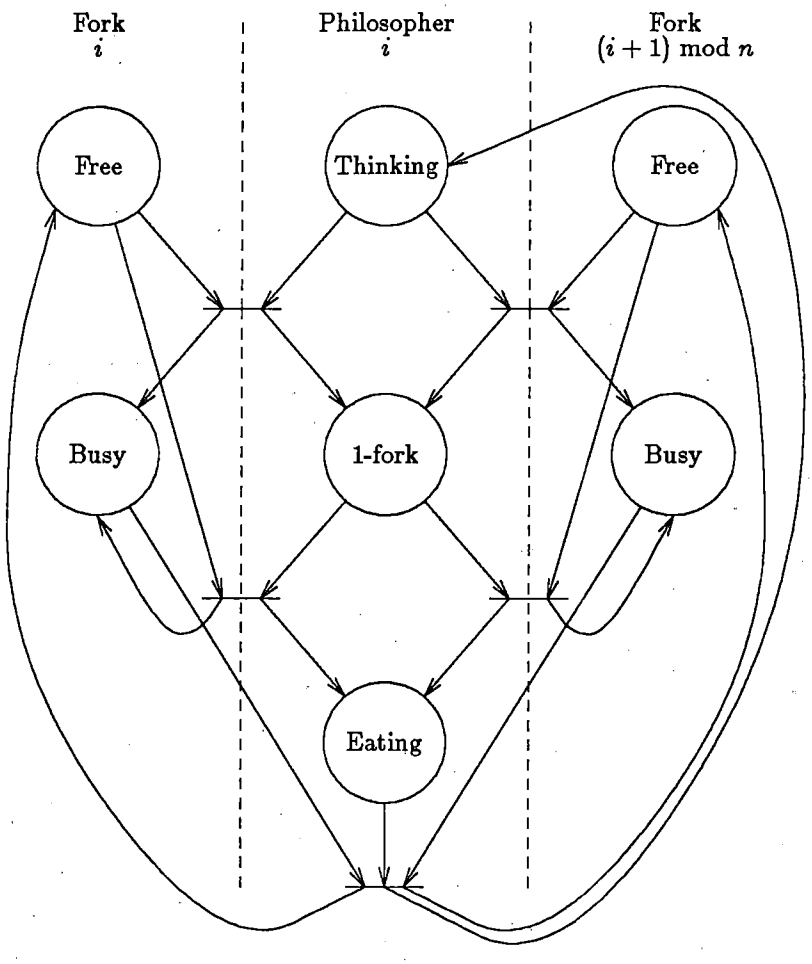

Figure 1

Petri Net for Dining Philosopher i

3. `exists s in S [p1_eating(s) > 0]`
   In the context of the dining philosopher example (discussed below), this predicate determines if it is possible for philosopher 1 to eat. It should be noted that this predicate does not in any way guarantee that philosopher 1 will ever eat.

2.5. A Simple Example

Now a short example is used to demonstrate some of the capabilities of the analyzer. The dining philosophers problem will be used because it is widely understood. Figure 1 shows the Petri net representation of one philosopher in the

```
/* Philosopher 1 model */
fork1_free,p1_thinking -> fork1_busy,p1_1_fork
fork2_free,p1_thinking -> fork2_busy,p1_1_fork
fork1_free,p1_1_fork -> fork1_busy,p1_eating
fork2_free,p1_1_fork -> fork2_busy,p1_eating
p1_eating,fork1_busy,fork2_busy -> fork1_free,fork2_free,p1_thinking

/* Philosopher 2 model */
fork2_free,p2_thinking -> fork2_busy,p2_1_fork
fork3_free,p2_thinking -> fork3_busy,p2_1_fork
fork2_free,p2_1_fork -> fork2_busy,p2_eating
fork3_free,p2_1_fork -> fork3_busy,p2_eating
p2_eating,fork2_busy,fork3_busy -> fork2_free,fork3_free,p2_thinking

/* Philosopher 3 model */
fork3_free,p3_thinking -> fork3_busy,p3_1_fork
fork1_free,p3_thinking -> fork1_busy,p3_1_fork
fork3_free,p3_1_fork -> fork3_busy,p3_eating
fork1_free,p3_1_fork -> fork1_busy,p3_eating
p3_eating,fork3_busy,fork1_busy -> fork3_free,fork1_free,p3_thinking

/* Initial state */
<fork1_free(1), fork2_free(1), fork3_free(1), p1_thinking(1),
 p2_thinking(1), p3_thinking(1)>
```

Figure 2

Dining Philosophers Problem for Three Philosophers

Symbolic Form of Petri Net

dining philosophers problem. The textual representation of the Petri net for the problem with three philosophers is shown in Figure 2. Using the P-NUT tools, this representation is converted into a canonical representation of the same net, from which the reachability graph is built. The reachability graph is then read by RGA so that the analysis can be done.

Figure 3 shows a sample run of the analyzer on the philosophers problem. The line numbers at the left have been added for reference purposes. In line 3, the user asks how many states there are in the reachability graph by determining the cardinality of the set of all states S. There are 26 states.

```
 1  Loading graph dining3.rg
 2  graph loaded
 3  >card(S)
 4  26
 5  >forall s in S [nsucc(s) > 0]              /* test for deadlock-freeness */
 6  false
 7  >{s in S | nsucc(s) = 0}                   /* find deadlocked state */
 8  {#22}
 9  >showstate(#22)                            /* print state symbolically */
10  fork1_busy p1_1_fork fork2_busy p2_1_fork fork3_busy p3_1_fork
11  >forall s in S [forall p in P [p(s) <= 1]]  /* test for safeness */
12  true
13  >forall s in S [marked(s) = tokens(s)]
14  true
15  >exists s in S [p1_eating(s) > 0]           /* can phil. 1 eat? */
16  true
17  >{s in S | p1_eating(s) > 0}
18  {#7, #19, #20}
19  > /* test that at most one philosopher can eat at once */
20  >forall s in S [p1_eating(s) + p2_eating(s) + p3_eating(s) <= 3/2]
21  true
```

Figure 3

Analysis of Dining Philosophers Problem

On line 5, the user asks if the net is deadlock free (that each state has at least one successor). RGA responds with `false`, that there is at least one deadlocked state. The user then asks on line 7 for the set of all states `s` which have no successors. The response is a set containing one state, #22. On line 9, the user asks for a symbolic display of state #22, with the result shown on line 10. The names are the names of the places as defined in the original Petri net, which have one token. Multiple tokens would have been indicated by an integer in parentheses following the place name.

In lines 11 and 13, the user determines if the net is safe, using two different methods. Line 11 uses the strict definition of net safety, while the expression on line 13 takes advantage of two built-in functions. Both expressions evaluate to `true`, indicating that the net is safe. The expression in line 13 is computed many times

faster than the one in line 11 because it avoids the doubly nested loops by using built-in functions.

All of the above tests apply equally to any net which might be analyzed with RGA. Next, some specific properties of the dining philosophers problem are analyzed. In line 15, the user asks if it is possible for philosopher 1 to eat, and the system responds `true`. The user then asks for the set of all states in which the philosopher is eating, and RGA responds with a set of three states. Finally, on line 20, the user verifies a property of the net which should be true, that the maximum number of philosophers eating in any state must be less than or equal to the number of philosophers divided by the number of forks needed by a philosopher to eat. If this expression did not evaluate to `true`, then an error in the specification of the net would be indicated. In this case, since integer division is used, $3/2$ is truncated to 1, and therefore no two philosophers can ever be eating simultaneously. In general, $\lfloor n/2 \rfloor$ philosophers can eat simultaneously when there are n dining philosophers.

3. Advanced capabilities

The features of RGA described above allow a designer to traverse the graph interactively and to prove some properties about the net. It may be possible, in some cases, for a designer to develop an algorithm to perform more complex analysis of the graph. It is therefore desirable to have the tool be able to execute user-defined algorithms built from the primitive capabilities outlined earlier.

RGA allows any value to be assigned to an identifier using the assignment operator (`:=`). It assigns the value of the expression on its right to the identifier on its left. In addition, it returns that value as a result. The sequence of expressions `a := 1` and `a := a + 1` would assign the identifier `a` the integer value 2. Then the

expression

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

would assign to a the set of all deadlocked states. Each identifier represents a <*value, type*> pair, so its type as well as its value can vary dynamically.

It is possible to assign to an identifier an expression, rather than the *value* of the expression, using the "::=" operator. The syntax used to define a function is

id (*formal-parameters*) [*locals-variables*] ::= *expression*

The *formal-parameters* and *locals-variables* are lists of identifiers separated with commas. If either of these lists is empty, then the corresponding parentheses or brackets are omitted. The function invocation mechanism in RGA provides for dynamic scoping of identifiers, as in LISP. Also as in pure LISP, recursion is often the primary mechanism for specifying interation.

In order that user-defined functions can be sufficiently powerful, two special expressions are included in the RGA language: expression lists and conditional expressions. The semicolon (;) infix operator evaluates the expression on its left and discards it, then evaluates and returns the expression on its right. It is an associative operator, so the expression "1;2;3" evaluates to the integer 3. Intuitively, it provides for sequential execution of expressions much like conventional programming languages. The `if` expression is used for conditional expression evaluation. It can take two different forms:

`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.

Figure 4 shows a user-defined function, `can_reach`. It constructs the set of states which can reach a particular state; thus it is the same as the primitive

```
EMPTYSET := {};

can_reach (s) ::= cr ({s}, EMPTYSET, {s})

cr(frontier, tried, canreachset)[nfrontier] ::=       \
   if frontier = EMPTYSET  \
      then canreachset  \
      else nfrontier := EMPTYSET;  \
           tried := union (tried, frontier); \
           forall s in frontier \
             [nfrontier := union(nfrontier,pred(s));  \
              canreachset := union (canreachset, pred(s)) ;  \
              true ] ; \
           cr(setdiff(nfrontier, tried), tried, canreachset) \
   fi
```

Figure 4

A Small User-Defined Function

operation `allpred` except that the constructed set will contain the initial state. With the dining philosophers problem, varying the number of philosophers from 2 to 8, it has been found that the primitive function is about 2.5 times faster than the user-defined version.

The \ symbols in the figure are used to indicate that the end of a line is not the end of the function definition. The `can_reach` function invokes a recursive function `cr` which does the actual work. `Cr` takes three arguments, a frontier of states which should be tried next, the set of states already tested, and the states which have already been found to be able to reach the starting state. If there are no more states in the frontier, then `cr` returns the set `canreachset` since nothing more can be added to that set. Otherwise, it adds the frontier set to the set of states which have been tried, and constructs the new frontier `nfrontier` as the set of predecessors of the states in the current frontier. Then `cr` is called recursively, removing any states from the new frontier which have already been tried.

4. An Extensive Example

4.1. The Alternating-Bit Protocol

A larger example is now presented which makes use of some of the more sophisticated features of RGA. The example used is the alternating-bit protocol [1], with message and acknowledgement queues of length two. In this protocol, there are two communicating entities, a sender and a receiver. The sender sends message packets to the receiver over an unreliable medium, with a flag bit attached to each message. This flag is either a zero or one, alternating from one message to the next. The receiver sends acknowledgement packets back to the sender which have the same flag bit as the message packet being acknowledged. If the receiver receives a packet with what it considers a bad flag, it still sends an acknowledgement packet containing the flag received, which then serves as a negative acknowledgement to the sender. The receiver then drops the bad packet.

Upon receiving a good packet, however, the receiver alternates its flag to be ready to receive the next packet from the sender. The sender alternates its flag and transmits its next message upon receiving a valid acknowledgment packet. It uses timeouts and negative acknowledgments to determine when it should stop waiting for an acknowledgement and retransmit its current message.

Figure 5 shows a Petri net representing a high-level description of of the alternating-bit protocol. For brevity, the global state of the system is represented as two digits, the first being the sender's flag, and the second being the receiver's flag. The *system* stays in state 00 until a message is successfully received by the receiver, and it then enters state 01. When the acknowledgement is correctly received by the sender, the system enters state 11; the sender has now alternated its flag, and is attempting to send the next message. When that message is correctly received, the system enters state 10, and after its acknowledgement is received by the sender, the system returns to state 00. The transitions labelled "Bad Message" and "Bad

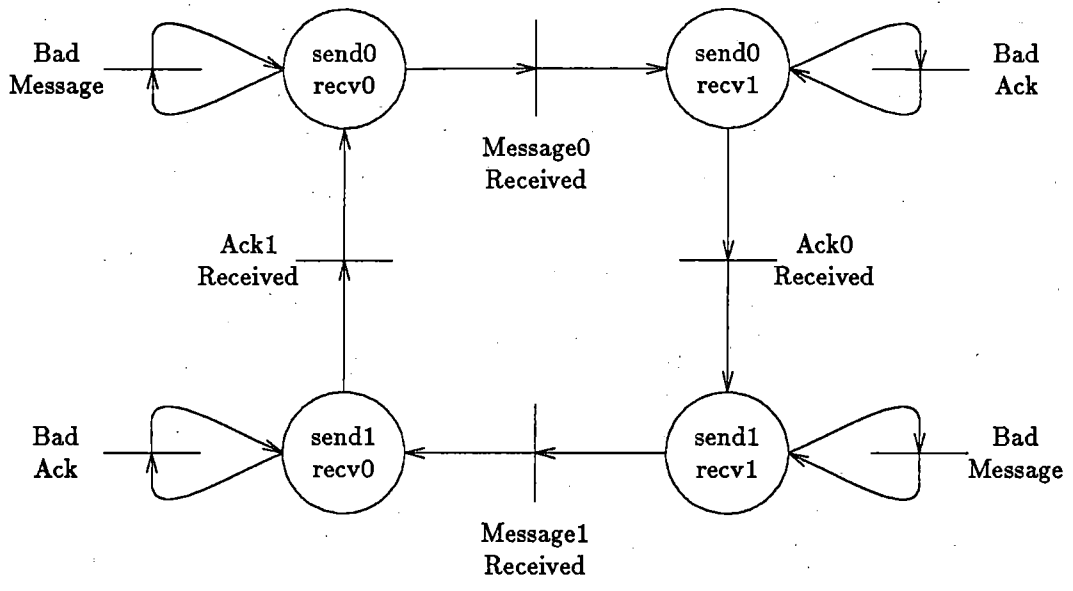

Figure 5

Meta Level Petri Net for Alternating-Bit Protocol

Ack" model timeouts and actions taken upon receipt of packets with bad flags. No other transitions should be possible in a correct implementation of the protocol.

4.2. Petri Net Model of the Alternating-Bit Protocol

Figures 6 through 8 show the textual representation of a detailed Petri net model of the alternating-bit protocol. The queues for messages being sent, and for the acknowledgements returning, are modeled as circular queues. Both queues are of length two in this model. The reachability graph for this Petri net has 1752 states.

The sender, shown in Figure 6, can be in one of four states: `Sready`, `waitack`, `ack0`, or `ack1`. The `send_flag0` and `send_flag1` places indicate the state of the sender's flag. If there is no room in the message queue when the sender is ready to send, then the sender will block until a slot becomes free. Once it has sent a message, it enters the `waitack` state to await the acknowledgement message from the receiver. Since simple Petri nets are incapable of representing time, the sender can retransmit

```
/* Sender sends msg */
Sready, send_flag0, slot0_empty, last0 -> msg0_0, last1, slot0_filled,
        send_flag0, waitack
Sready, send_flag1, slot0_empty, last0 -> msg1_0, last1, slot0_filled,
        send_flag1, waitack
Sready, send_flag0, slot1_empty, last1 -> msg0_1, last0, slot1_filled,
        send_flag0, waitack
Sready, send_flag1, slot1_empty, last1 -> msg1_1, last0, slot1_filled,
        send_flag1, waitack

/* Sender dequeues an acknowledgement packet. */
waitack, first_ack0, filled_ack0, ack_msg0_0 -> first_ack1, ack_slot0_empty,
        ack0
waitack, first_ack0, filled_ack0, ack_msg1_0 -> first_ack1, ack_slot0_empty,
        ack1
waitack, first_ack1, filled_ack1, ack_msg0_1 -> first_ack0, ack_slot1_empty,
        ack0
waitack, first_ack1, filled_ack1, ack_msg1_1 -> first_ack0, ack_slot1_empty,
        ack1

/* Good ack - alternate bit and send next message */
ack0, send_flag0 -> send_flag1, Sready
ack1, send_flag1 -> send_flag0, Sready

/* Bad ack - ignore it and let timeout take care of retransmitting msg */
ack0, send_flag1 -> send_flag1, waitack
ack1, send_flag0 -> send_flag0, waitack

/* Timeout (if no acknowledgements available) and retransmit last msg */
waitack, first_ack0, ack_slot0_empty -> Sready, first_ack0, ack_slot0_empty
waitack, first_ack1, ack_slot1_empty -> Sready, first_ack1, ack_slot1_empty
```

Figure 6

Alternating Bit Protocol — Sender Model

its last message at any time when it is waiting for an acknowledgement and there are no acknowledgements in the queue.

The receiver, Figure 7, is in one of four states, `Rready`, `acking`, `read0`, and `read1`. When `Rready`, the receiver waits until a message appears in the message queue from the sender. When a message comes in, it verifies that the flag bit on the message corresponds to the type of message expected (`rcv_flag0` or `rcv_flag1`). If the flags match, an acknowledgement is sent with the same flag bit, and the receiver's flag is reversed in preparation for the next message. Otherwise, if the flag

```
/* Receiver dequeues msg */
Rready, slot0_filled, first0, msg0_0 -> read0, first1, slot0_empty
Rready, slot0_filled, first0, msg1_0 -> read1, first1, slot0_empty
Rready, slot1_filled, first1, msg0_1 -> read0, first0, slot1_empty
Rready, slot1_filled, first1, msg1_1 -> read1, first0, slot1_empty

/* Receiver verifies received msg matches rcv_flag */
read0, rcv_flag0 -> rcv_flag1, acking, msg0  /* Flip rcv_flag if good msg */
read1, rcv_flag1 -> rcv_flag0, acking, msg1
read0, rcv_flag1 -> rcv_flag1, acking, msg0  /* Don't flip flag on bad msg */
read1, rcv_flag0 -> rcv_flag0, acking, msg1

/* Receiver sends an acknowledgement with flag = flag received */
acking, msg0, ack_slot0_empty, last_ack0 -> Rready, ack_msg0_0, last_ack1,
        filled_ack0
acking, msg1, ack_slot0_empty, last_ack0 -> Rready, ack_msg1_0, last_ack1,
        filled_ack0
acking, msg0, ack_slot1_empty, last_ack1 -> Rready, ack_msg0_1, last_ack0,
        filled_ack1
acking, msg1, ack_slot1_empty, last_ack1 -> Rready, ack_msg1_1, last_ack0,
        filled_ack1
```

Figure 7

Alternating Bit Protocol — Receiver Model

```
/* Discard messages at random from both queues */
slot0_filled, first0, msg0_0 -> slot0_empty, first1
slot0_filled, first0, msg1_0 -> slot0_empty, first1
slot1_filled, first1, msg0_1 -> slot1_empty, first0
slot1_filled, first1, msg1_1 -> slot1_empty, first0

filled_ack0, first_ack0, ack_msg0_0 -> ack_slot0_empty, first_ack1
filled_ack0, first_ack0, ack_msg1_0 -> ack_slot0_empty, first_ack1
filled_ack1, first_ack1, ack_msg0_1 -> ack_slot1_empty, first_ack0
filled_ack1, first_ack1, ack_msg1_1 -> ack_slot1_empty, first_ack0

/* INITIAL CONDITIONS */
<Sready, send_flag0, Rready, rcv_flag0, slot0_empty, slot1_empty,
 ack_slot0_empty, ack_slot1_empty, last0, last_ack0, first0, first_ack0>
```

Figure 8

Alternating Bit Protocol — Line Noise and Initial State

bits did not match, the receiver sends an acknowledgement with a flag bit indicating the type of message it received, giving the sender a negative acknowledgement.

Figure 8 shows the model of transmission line noise and the initial state of the Petri net. This model of the alternating-bit protocol assumes that there is some probability that line noise will destroy messages or acknowledgements. The model never distorts messages (changes the message sequence number). There are therefore some transitions which can absorb messages or acknowledgements before they are received, simply dropping the packets. In the initial state, the sender is ready to send a message with flag bit 0, the receiver is ready to receive a message with flag bit 0, and both the message and acknowledgement queues are empty.

4.3. Verification of the Model

We have found that RGA can be useful in verifying that a Petri net model of a system is correct, as well as in analyzing the system being modeled. Our understanding of the alternating-bit protocol allows us to state several properties this model of the protocol should exhibit if it is correct:

1. The model should be safe since each place is used as a boolean flag.

2. The sum of the tokens on `Sready`, `waitack`, `ack0`, and `ack1` will be 1 for all states since these are mutually exclusive conditions. `Send_flag0` and `send_flag1` are also mutually exclusive.

3. The sum of the tokens on `Rready`, `acking`, `read0`, and `read1` should be 1 for all states. This also verifies that `read0` and `read1` are mutually exclusive.

4. The sum of `rcv_flag0` and `rcv_flag1` should be 1 for all states, verifying that the receiver expects only one type of message at any one time.

5. For each slot in the each queue, at most one flag bit should be set. For instance, `msg0_0` and `msg1_0` should be mutually exclusive. In states where neither are set, then that slot should be available (e.g., `slot0_empty`). A slot should not be both available and filled (e.g., `slot0_empty` and `slot0_filled` are mutually exclusive).

6. The system should behave as described by Figure 5. Transition-firings should not exist between the states other than those shown. The transitions which loop from each of the system states should be fired only when a message or acknowledgement has *not* been received successfully. To get from state 00 to state 11, or state 11 to state 00, exactly one good message should have

```
/* Test for safeness */
is_safe ::= forall s in S [tokens(s) = marked(s)]

/* Test for consistency of the sender */
sender_consistent ::= forall s in S [(Sready + waitack + ack0 + ack1 = 1) & \
                                     (send_flag0 + send_flag1 = 1) ]

/* Test receiver properties */
rcvr_consistent ::= forall s in S [(Rready + acking + read0 + read1 = 1) & \
                                   (rcv_flag0 + rcv_flag1 = 1) ]

/* Test consistency of the message and acknowledgement queues */
qs_consistent ::= \
    forall s in S [slot0_filled + slot0_empty = 1 & \
                   slot1_filled + slot1_empty = 1 & \
                   filled_ack0 + ack_slot0_empty = 1 & \
                   filled_ack1 + ack_slot1_empty = 1] & \
    forall s in S [(msg0_0 + msg1_0=0 iff slot0_empty) & \
                   (msg0_1 + msg1_1=0 iff slot1_empty) & \
                   (ack_msg0_0 + ack_msg1_0=0 iff ack_slot0_empty) &\
                   (ack_msg0_1 + ack_msg1_1=0 iff ack_slot1_empty)]&\
    forall s in S [msg0_0 + msg1_0 <= 1 & msg0_1 + msg1_1 <= 1 & \
                   ack_msg0_0 + ack_msg1_0 <= 1 & ack_msg0_1 + ack_msg1_1 <= 1]
```

Figure 9

Verification of Alternating-Bit Protocol Model

been received by the receiver, and one good acknowledgement received by the sender.

Properties 1 through 5 can be seen as properties which are used to verify that the model behaves in a way which is consistent with our intent. Property 6 verifies that it behaves as a correct alternating-bit protocol.

Figure 9 shows the RGA functions used to verify that these properties hold for this model. The test for net safeness is discussed above. The `sender_consistent` function tests that `waitack` and `Sready` are mutually exclusive, and that the sender is always in one of those two states or in the intermediate state of deciding whether it has received a valid acknowledgement (`ack0` or `ack1`). It then tests that for states which have a token on `send_flag0` or `send_flag1`, there are never tokens on both places at once.

The receiver is consistent if it is waiting to read, has read a packet with a zero or one flag, or it is sending an acknowledgement with a zero or one flag. These states are all mutually exclusive. The receiver must in any state be prepared for the next packet read to have a specific flag value, so `rcv_flag0(s) + rcv_flag1(s)` must also be one for all states `s`. The `rcvr_consistent` function tests for all of these conditions simultaneously.

The sender and acknowledgement queues are in consistent states if the slots are always filled or empty, there is not a message in a slot if and only if the slot is empty, and there is never more than one message type in any slot. These conditions are tested by function `qs_consistent`.

Finally, we wish to show that the system exhibits the overall flow described in Figure 5. The variables `s0r0`, `s0r1`, `s1r0`, and `s1r1`, initialized in the `make_subsets` routine, are the mutually-exclusive subsets of all states of the system based on the marking of the `send_flag0`, `send_flag1`, `recv_flag0`, and `recv_flag1` places. They represent the four places in Figure 5. The sets of transitions `tr0r1` and `tr1r0` are the transitions which are fired upon the receipt of a valid message packet, while `ts0s1` and `ts1s0` are those which are fired upon receipt of acknowledgment packets. In Figure 6, `ts0s1` is the ninth transition, and `ts1s0` is the tenth. In Figure 7, `tr0r1` is the fifth transition shown, and `tr1r0` is the sixth.

The `subset_ok` function tests that from one set of states, all transitions lead to either that same or to the subsequent place (set of states) in the net of Figure 5. The sets `s1` and `s2` are the sets of states which represent consecutive places in the net of Figure 5. For example, `s0r0` and `s0r1`. The set `t` is the set of transitions fired upon a successful receipt of a message or acknowledgement (e.g., `tr0r1`). There will be at least one of each of these types of transitions. Then all the transitions out of the place are tested to verify that they lead to either the same or the subsequent set of states. In addition, a transition from a set of states to itself (e.g., `s0r0` to

```
/* Test that the system exhibits expected overall flow */

/* Create subsets of states used in functions below */
s0r0 := {s in S | send_flag0 & rcv_flag0};
s0r1 := {s in S | send_flag0 & rcv_flag1};
s1r0 := {s in S | send_flag1 & rcv_flag0};
s1r1 := {s in S | send_flag1 & rcv_flag1};
s0 := {s in S | send_flag0}; s1 := {s in S | send_flag1};
r0 := {s in S | rcv_flag0}; r1 := {s in S | rcv_flag1};
tr0r1 := setop(trans, intersection(setop(tfout, r0), setop(tfin, r1)));
tr1r0 := setop(trans, intersection(setop(tfout, r1), setop(tfin, r0)));
ts0s1 := setop(trans, intersection(setop(tfout, s0), setop(tfin, s1)));
ts1s0 := setop(trans, intersection(setop(tfout, s1), setop(tfin, s0)));

card(s0r0) > 0 & card(s0r1) > 0 & card(s1r0) > 0 & card(s1r1) > 0 & \
    card(tr0r1) = 1 & card(tr1r0) = 1 & card(ts0s1) = 1 & card(ts1s0) = 1

/* A subset s1 is "ok" if all transition-firings lead to other states in the
   same group or to states in the next metastate */
subset_ok(s1, s2, t)[tfs, tf, s] ::= \
    tfs := setop(tfout, s1); \
   (exists tf in tfs [in(dest(tf), s1)] & \
    exists tf in tfs [in(dest(tf), s2)] & \
    forall tf in tfs [s := dest(tf); \
        ((in(s, s1) & not in(trans(tf),t)) | (in(s, s2) & in(trans(tf),t)))] )

/* The model is correct if each metastate is "ok" by the above definition */
subset_ok(s0r0, s0r1, tr0r1) & subset_ok(s0r1, s1r1, ts0s1) & \
    subset_ok(s1r1, s1r0, tr1r0) & subset_ok(s1r0, s0r0, ts1s0)
```

Figure 10

Verification of Overall Model Behavior

s0r0) must not involve firing the successful-receipt transition, while those leading to the second state (s0r1) must always do so.

The subsets_ok function merely invokes subset_ok for each of the four pairs of places in Figure 5 and the appropriate set of transitions. Since this function returns true, RGA verifies that the model does exhibit the behavior expected of the alternating-bit protocol. It also verifies that in going from state 00 to state 11, or from state 11 to state 10, exactly one message and one acknowledgement are successfully received.

5. Conclusion

The reachability graph analyzer program allows many system-independent and system-specific properties of concurrent systems modeled using Petri nets to be verified. The tool provides primitives from which complex user-defined propositions can be constructed about the states of the reachability graph. RGA then mechanically verifies the propositions for a particular graph. It has been used to analyze graphs with over 6500 states. Often, clever function definitions can be used to avoid brute-force approaches to verifying system properties, leading to significant time savings with large graphs.

Other tools are needed to aid in the automatic generation and analysis of Petri nets. RGA is only one of a suite of tools currently being developed for this purpose at UCI. Other tools include Petri net editors, optimizers, animators, and simulators, and reachability and decision graph builders. The tools are being designed to interconnect easily, allowing the greatest possible flexibility in the design and analysis of concurrent systems expressed as Petri nets.

REFERENCES

[1] K. A. Bartlett, R. A. Scantlebury, and P. T. A. Wilkinson, "Note on Reliable Full-Duplex Transmission Over Half-Duplex Links," *CACM*, vol. 12, no. 5, pp. 260–261, May, 1969.

[2] G. Berthelot and R. Terrat, "Petri Net Theory for the Correctness of Protocols," *Proceedings, Second International Workshop on Protocol Specification, Testing, and Verification*, C. A. Sunshine, Ed., pp. 325–342, Amsterdam: North-Holland, May, 1982.

[3] K. Gostelow, V. G. Cerf, G. Estrin, and S. Volansky, "Proper Termination of Flow of Control in Programs Involving Concurrent Processes," *SIGPLAN Notices*, vol. 7, no. 11, 1972.

[4] B. Hailpern and S. Owicki, "Verifying Network Protocols Using Temporal Logic," *Proc. of Trends and Applications Symposium, 1980, Computer Network Protocols*, Maryland: National Bureau of Standards, May, 1980.

[5] R. M. Keller, "Formal Verification of Parallel Programs," *Commun. ACM*, vol. 19, no. 7, pp. 371–384, July, 1976.

[6] J. Martìnez and M. Silva, "A Simple and Fast Algorithm to Obtain All Invariants of a Generalized Petri Net," *Second European Workshop on Application and Theory of Petri Nets*, C. Girault and W. Reisig, Eds., pp. 301–310, New York: Springer-Verlag, 1982.

[7] E. T. Morgan, "RGA Users Manual," Technical Report Number 243, University of California, Irvine: Department of Information and Computer Science, Dec., 1984.

[8] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs: Prentice-Hall, 1981.

[9] J. Postel, "Graph Modeling of Computer Communications Protocols," *Proceedings of the 5th Texas Conference on Computing Systems*, pp. 66–77, Austin, Texas, Oct., 1976.

[10] R. R. Razouk and G. Estrin, "Modeling and Verification of Communication Protocols: The X.21 Interface," *IEEE Transactions on Computers*, vol. C-29, no. 12, pp. 1038–1052, Dec., 1980.

[11] R. R. Razouk and C. V. Phelps, "Performance Analysis Using Timed Petri Nets," Technical Report Number 206, University of California, Irvine: Department of Information and Computer Science, Aug., 1983.

[12] K. Sabnani and M. Schwartz, "Verification of a Multidestination Protocol Using Temporal Logic," *Protocol Specification, Testing, and Verfication*, C. A. Sunshine, Ed., Amsterdam: North-Holland, 1982.

[13] J. Sifakis, "Petri Nets for Performance Evaluation," *Measuring, Modelling, and Evaluating Computer Systems*, H. Beilner and E. Gelenbe, Eds., pp. 75–93, Amsterdam: North-Holland, 1977.

[14] D. M. Stanat and D. F. McAllister, *Discrete Mathematics in Computer Science*, Englewood Cliffs: Prentice-Hall, 1977.

[15] C. A. Sunshine, "Formal Techniques for Protocol Specification and Verification," *IEEE Computer*, pp. 20–27, Sept., 1979.

[16] F. J. W. Symons, "Verification of Communication Protocols Using Numerical Petri Nets," *Australian Telecommunication Research*, vol. 14, no. 1, pp. 34–38, 1980.

[17] W. M. Zuberek, "Timed Petri Nets and Preliminary Performance Evaluation," *7th Annual Symposium on Computer Architecture*, pp. 88–96, 1980.