Petri Net Equivalence

Richard D. Sidwell

Technical Report 87-02
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

ABSTRACT

Determining whether two Petri nets are equivalent is an interesting problem from both practical and theoretical standpoints. Although it is undecidable in the general case, for many interesting nets the equivalence problem is solvable. This paper explores, mostly from a theoretical point of view, some of the issues of Petri net equivalence, including both reachability sets and languages. Some new definitions of reachability set equivalence are described which allow the markings of some places to be treated identically or ignored, analogous to the Petri net languages in which multiple transitions may be labeled with the same symbol or with the empty string. The complexity of some decidable Petri net equivalence problems is analyzed.

This work was supported in part by NSF grant DCR 84-06756

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

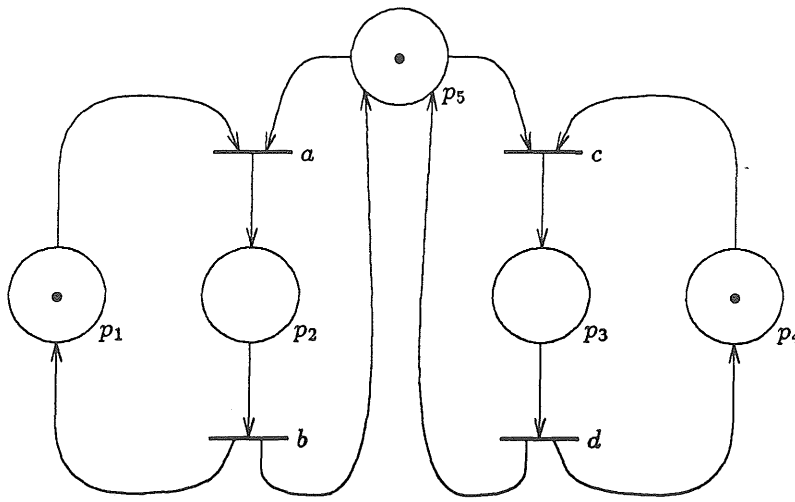

Figure 1
A simple Petri net

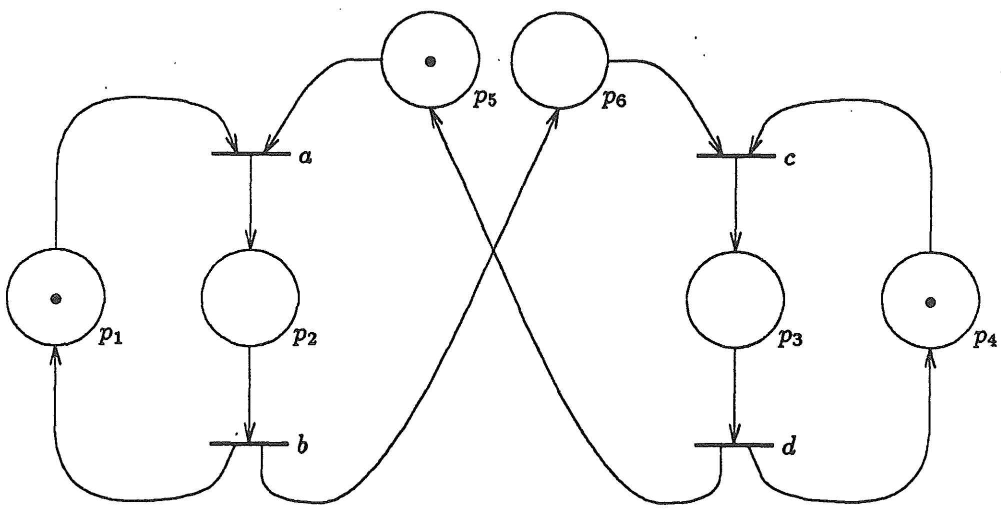

Figure 2
A Petri net with the same reachability set as Figure 1

by Hack in his work ([Hack76a], [Hack76b]), and the languages P^f, P, and P^λ described in [Pete81].

As an example of the independence of reachability set equivalence and language equivalence, consider the Petri nets shown in Figures 1–4. Figure 1 shows a simple Petri net which implements mutual exclusion; a token will never be in places p_2 and p_3 at the same time. Figure 2 shows a Petri net with the same reachability set as the one in Figure 1 (assuming the homomorphisms map places p_5 and p_6 in net 2 to the same tuple-position as place p_5 in net 1). However, the transition firing sequences are different since a and b must fire before c can in Figure 2, but not in

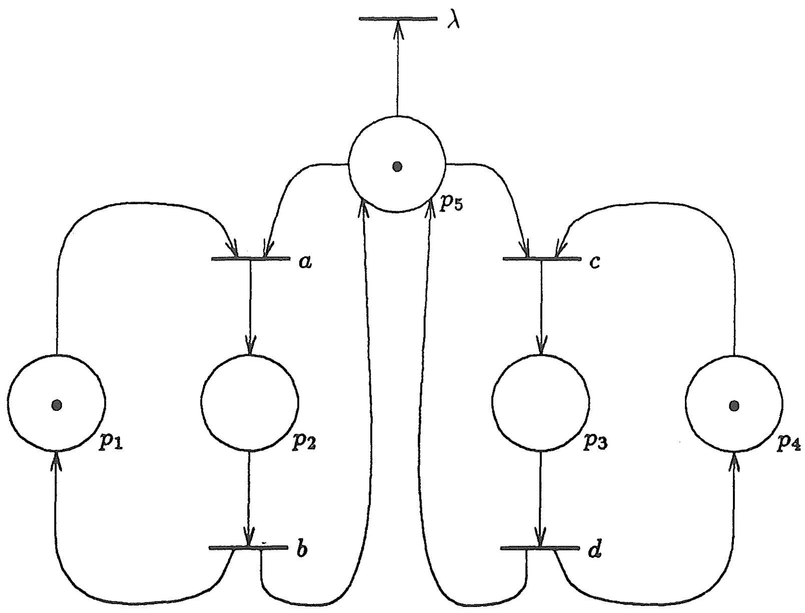

Figure 3
A Petri net with the same language as Figure 1

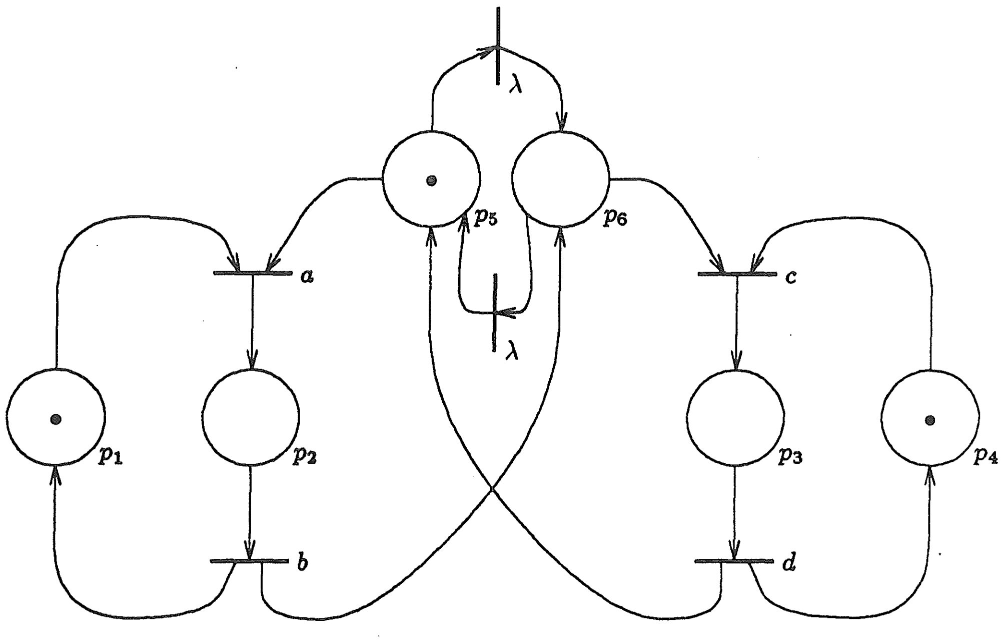

Figure 4
A Petri net isomorphic to that in Figure 1

Figure 1. Figure 3 is a Petri net with the same language as Figure 1, but different reachability sets since the top transition labeled λ can fire without affecting the language. Figure 4 is a Petri net with both the same reachability set and the same language as that of Figure 1.

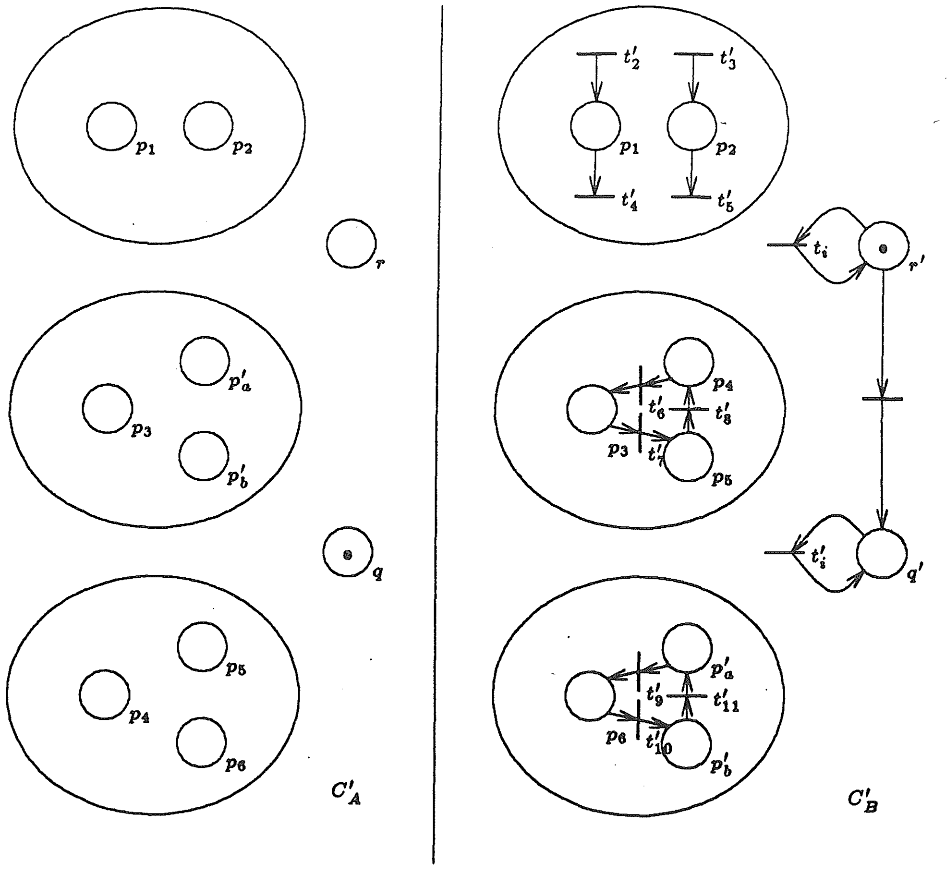

Figure 5
Reducing $\mathcal{R}^\lambda$ to $\mathcal{R}^f$

Relationship of Different Equivalence Definitions

The different ways of defining Petri net equivalence are independent to some extent, but are related nevertheless. Obviously, equivalence for $\mathcal{R}^f$ and $\mathcal{R}$ are reducible to equivalence for $\mathcal{R}^\lambda$. It is not too difficult to reduce $\mathcal{R}^\lambda$ to R^f in linear time and space (the reduction is given below), so the complexity for solving any of the reachability set equivalence problems should be about the same. The analogous reductions are not possible for languages; equivalence for $\mathcal{L}^f$ and $\mathcal{L}$ are obviously reducible to $\mathcal{L}^\lambda$, but $\mathcal{L}^f$ is easier to solve than equivalence for $\mathcal{L}$ or $\mathcal{L}^\lambda$. This fact was proven in [HACK76A], where it was demonstrated that equality for $\mathcal{L}$ and $\mathcal{L}^\lambda$ are undecidable, but equality for $\mathcal{L}^f$ is equivalent to the reachability problem, which was proven decidable in [MAYR81]. It is also possible to reduce equivalence for $\mathcal{R}^\lambda$ to equivalence for $\mathcal{L}^\lambda$ and vice versa (the reductions are given below). All these reductions can be done using linear space and time, so the complexity for solving any of the equivalence problems, except for $\mathcal{L}^f$, should be about the same.

The problem of equivalence for $\mathcal{R}^\lambda$ is reduced to equivalence for $\mathcal{R}^f$ using a construction similar to one used by Hack in reducing the polynomial graph inclusion problem to the subset problem for $\mathcal{R}^f$. Given two Petri nets, C_A and C_B, we construct nets C'_A and C'_B such that $R^\lambda(C_A, h_A) \subseteq R^\lambda(C_B, h_B)$ if and only if $R^f(C'_A, h'_A) \subseteq R^f(C'_B, h'_B)$ (see Figure 5). The subset problem is reducible to the equivalence problem as shown in [HACK76B] (also in [PETE81]). First, we make the number of places in the two Petri nets the same by adding, for each *equivalence set* (a set containing all the places in the net for which the values of $\hat{h}$ are equal), enough places to the proper net to make the size of the corresponding equivalence sets in the two nets equal. Next (Hack's contribution), we eliminate the effects of all *unobservable* places (those for which $\hat{h}(p) = 0$) by adding places q and r to C'_A, and adding places q' and r' to C'_B. In C'_A, r and q are not used for any transitions, and the initial marking includes one token in q and none in r. In C'_B, the place r' is a typical *run* place, which is initially marked, and is included as both an input and an output to each transition in the original net C_B. Thus, the net operates normally as long as a token remains in r', but if the token is removed the net *freezes*. The token is removed by adding a transition from r' to q'. For each place p_i for which $\hat{h}_B(P_i) = 0$ (including any places added to equalize the two nets), we add one transition with q' and p_i as inputs and q' as an output, and another transition with q' as an input and q' and p_i as outputs. These transitions allow each unobservable place to have as many (or as few) tokens as desired. Finally, we eliminate the effects of identical places by adding a transition between every pair of places p_i and p_j (including any places added to equalize the two nets) such that $\hat{h}_B(p_i) = \hat{h}_B(p_j)$ with inputs q' and p_i and outputs q' and p_j. This allows the tokens in any set of identical places to be arbitrarily distributed among the places in the set.

The equivalence problem for $\mathcal{R}^\lambda$ is reduced to the equivalence problem for $\mathcal{L}^\lambda$ by constructing new nets C'_A and C'_B such that $R^\lambda(C_A, h_A) = R^\lambda(C_B, h_B)$ if and only if $L^\lambda(C'_A, h'_A) = L^\lambda(C'_B, h'_B)$ (refer to Figure 6). To construct the Petri net C' from the Petri net C (for either C_A or C_B), add a run place r, initially marked with one token, which is an input and an output to each transition in C. Also add place q, initially unmarked, with a transition t_q from r to q. The net thus runs normally until t_q is fired, which freezes it. Add new transitions t_{p_i}, each with two inputs, p_i and q, and one output, q, for each place in C. The function $\hat{h}'$ supporting the new homomorphism h' is equal to λ for each transition in C' except for the transitions t_{p_i}, which have the value λ if $\hat{h}(p_i) = 0$ or the value $\hat{h}(p_i)$ otherwise (the alphabet Σ is the same as the range of $\hat{h}$ excluding 0).

The net C' operates as follows: Since r is initially marked, the net operates the same until an arbitrary reachable marking is reached, when t_q fires, freezing the part of the net that corresponds to the original net C, and enabling the new transitions t_{p_i}. Since all transitions fired up to this point have been labeled with λ, no string has yet been generated. With a token in q, however, each transition t_{p_i} can fire as many times as there are tokens in p_i, adding up to that many symbols somewhere in the language. The language generated by C' is thus

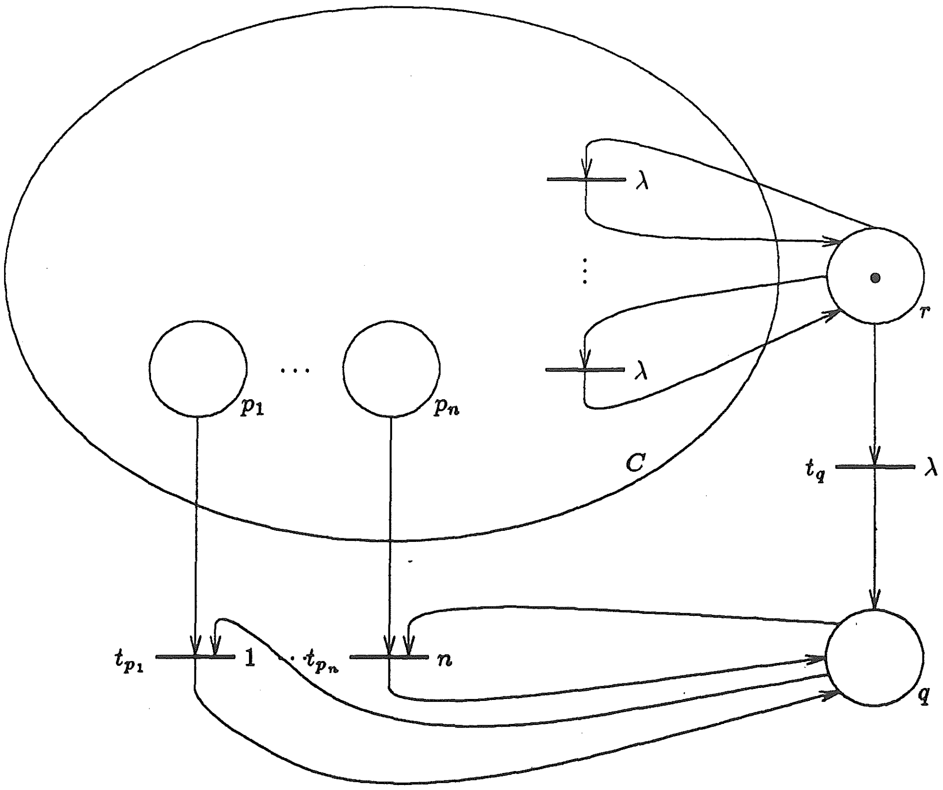

Figure 6
Reducing $\mathcal{R}^\lambda$ to $\mathcal{L}^\lambda$

$\{\, x \mid \psi(x) \leq h(\mu)$ for some $\mu \in R(C)\,\}$, where ψ is the Parikh mapping (a mapping from a string to the number of occurrences of each symbol in the string[1]).

The equivalence problem for $\mathcal{L}^\lambda$ is reduced to the equivalence problem for $\mathcal{R}^\lambda$ by constructing new nets C'_A and C'_B such that $L^\lambda(C_A, h_A) = L^\lambda(C_B, h_B)$ if and only if $R^\lambda(C'_A, h'_A) = R^\lambda(C'_B, h'_B)$. To construct the Petri net C' from the Petri net C, add a place r, initially marked with one token, which is an input (but not an output) to each transition in C. Add another place, c, initially marked with one token. The number of tokens on this place will be doubled (weakly) each time a transition of C fires. Add two more places, n and t, initially unmarked, and two more transitions t_r and t_n. Transition t_r has n as its only input, and r as its only output. Transition t_n has places n and t as inputs, and as outputs, one arc to n and two arcs to c. Finally, for each transition t_i in C, add place a_i as an output for t_i, place p_{t_i}, transition t_{a_i} with input places a_i and c, and outputs a_i, p_{t_i} and t, and transition t_{d_i}, with input place a_i and output place n. The function $\hat{h}'$ supporting the new homomorphism h' is equal to 0 for all places of C' except for places p_{t_i};

[1] For example, over the alphabet $\{\, a, b, c, d, r\,\}$, $\psi(abracadabra) = (5, 2, 1, 1, 2)$, since there are 5 a's, 2 b's, etc.

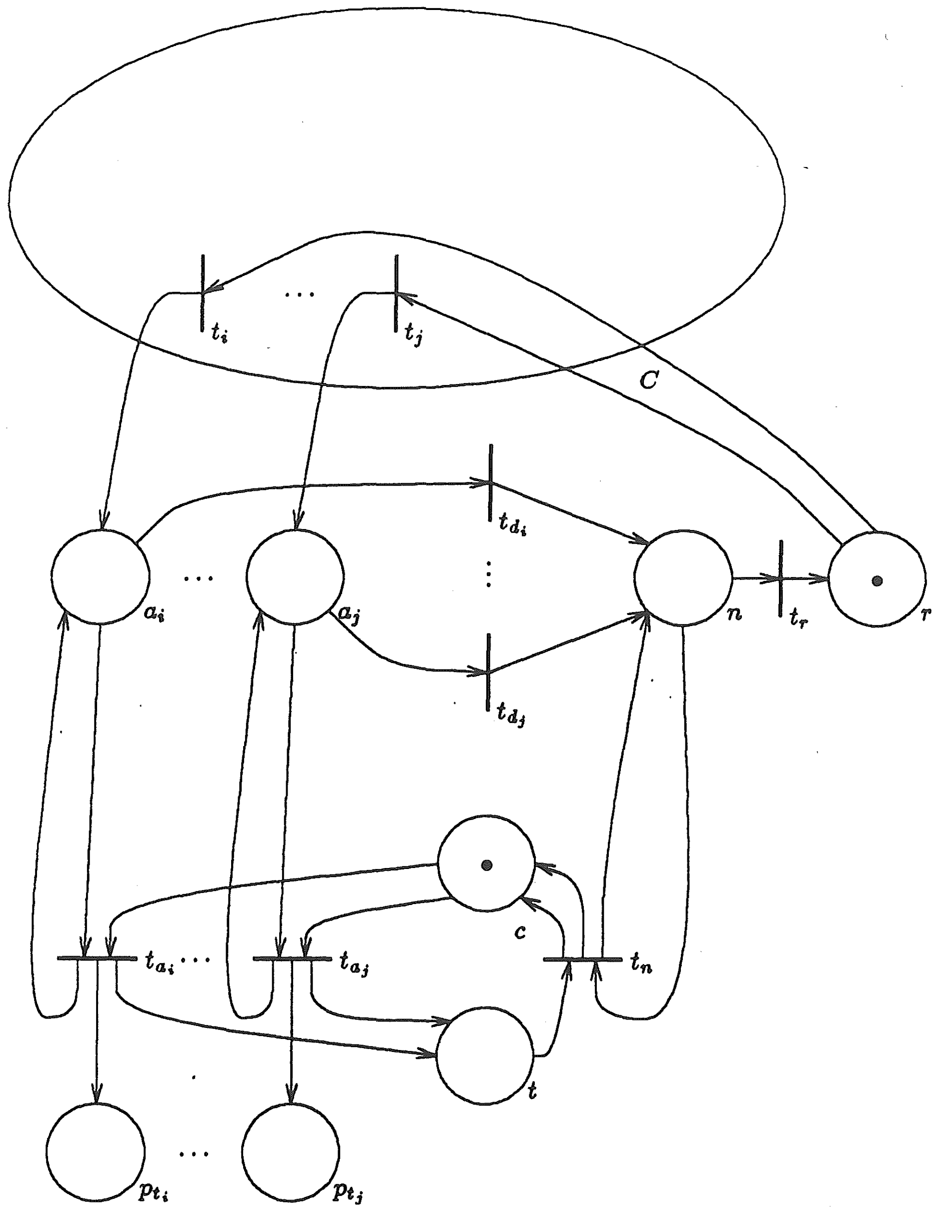

Figure 7
Reducing $\mathcal{L}^\lambda$ to $\mathcal{R}^\lambda$

for these places, let $\sigma(a)$ be the ordinal value of symbol a in an ordering σ of the alphabet Σ, with $\sigma(\lambda) = 0$. Then, $\hat{h}'(p_{t_i}) = \sigma(\hat{h}(t_i))$.

The new net C' works as follows (see Figure 7): When there is a token on r, any enabled transition of C can fire; assume that t_i does. Firing t_i removes the token from r and places a token on a_i, as well as making the changes to C' that would normally be made to C. With a token on a_i, transition t_{a_i} can now fire once for each token in c, moving the tokens from c to t, and weakly adding c to p_{t_i}. When all the tokens are gone from c (or sooner since we only add weakly), transition t_{d_i}

fires, placing a token in n, and enabling transition t_n, which can fire once for each token in t, and places twice this many tokens in c. When this is done (or sooner), transition t_r fires, placing a token in r again so that the cycle can repeat. At any time, place p_{t_i} contains between 0 and $\sum 2^j$ tokens, where $\beta_j = t_i$ in the firing sequence $\beta_1\beta_2\ldots\beta_k$. Thus, the possible markings for these places correspond to the possible transition firings of the original net, and the reachability sets of two nets constructed in this way will be equivalent if and only if the languages of the original nets are equivalent.

Unfortunately, these reductions are mostly of theoretical interest, and have little practical use; the equivalence problem for all these classes is undecidable in the general case. There are special instances of Petri nets where equivalence is decidable, but these reductions will not generally preserve the necessary restrictions. For example, reachability set equivalence is decidable for Petri nets with five places (see [HOPC79]), but language equivalence is not (see [VALK81, p 323]). The equivalence between $\mathcal{R}$ and $\mathcal{L}$ shown here does not apply since the reductions add places to the net, invalidating the restriction to five places.

Methods For Determining Equivalence

There are several general methods for determining whether two Petri nets have equivalent reachability sets or languages. If they are finite, the obvious way is to simply generate and compare them. Since they are infinite in the general case, we must use other methods. One method is to compare closed forms for them. For example, to compare two languages which happen to be regular, we can generate regular expressions that represent each language in a finite form and compare them. It is usually easier to compare them if they can be put into a canonical form.

Another method is to find a way to transform one Petri net into the other using transformations known to preserve equivalence. For example, it is obvious that replacing a place by two places connected together by a λ-labeled transition with the inputs of the original place being connected to the first place, and the outputs to the second will not change either the language or reachability set of the net. Other transformations may depend on certain properties of the net. For example, a dead transition, which can never be enabled, may be removed without affecting the net. This is the obvious method to use if the nets being compared were somehow derived from each other.

It is also possible to use a combination of these techniques. For example, some transformations may be applied to two Petri nets to simplify them, or put them into a form for which a closed form may more easily be generated, then the closed forms generated and compared.

The rest of the paper will discuss the complexity of determining Petri net equivalence for reachability sets and languages. Only comparison of closed forms will be considered; the study of Petri net transformations is another topic of Petri net research.

Reachability Set Equivalence

The reachability set equivalence problem for general Petri nets is undecidable (see [HACK76B] or [PETE81] for a proof of the undecidability of $\mathcal{R}^f$; the undecidability of $\mathcal{R}$ and $\mathcal{R}^\lambda$ follows immediately). For many special Petri nets however, equivalence can be decided. In general, the reachability sets of these Petri nets have a special property which allows them to be compared. For example, for bounded Petri nets, with finite reachability sets, equivalence is obviously decidable.

Another property that the reachability sets for many common classes of Petri nets have is that of being semilinear [GINS66]. A set $R \subseteq N^r$ is *linear* if there exist vectors $f_0, f_1, \ldots, f_n$ in N^r such that:

$$R = \{ f_0 + x_1 f_1 + \cdots + x_n f_n \mid x_1, \ldots, x_n \text{ are in } N \}.$$

A set R is *semilinear* if it is a finite union of linear sets. Semilinear sets correspond exactly to predicates expressible in Presburger arithmetic, for which procedures exist to decide ([GINS66, OPPE78]), thus reachability set equivalence is decidable for any type of Petri net that has a semilinear reachability set.

Hopcroft and Pansiot have shown that Petri nets with at most five places have semilinear reachability sets ([HOPC79]). A *reversible* Petri net is one in which for every transition there is another transition with inputs and outputs reversed, allowing any transition to be "undone" by firing its complementary transition; reversible Petri nets have semilinear reachability sets ([ARAK77]). A *persistent* Petri net is one in which any enabled transition is disabled only by firing it (i.e., firing a transition will not disable a different one); these nets also have semilinear reachability sets ([GRAB80]). A Petri net is *weakly persistent* if, whenever transition sequences β and β' are firable and β covers β', there exists a rearrangement $\beta'\beta''$ of β that is fireable; these, too, have semilinear reachability sets ([YAMA81]). Yamasaki defines a class of Petri nets called *normal* Petri nets which have semilinear reachability sets ([YAMA84]). A Petri net is normal if each transition t which has an input place in a minimal circuit c also has an output place in c.

All these classes (and probably many more) have semilinear reachability sets, and therefore the reachability set equivalence problem is decidable for them. However, just because a problem is decidable does not mean that it is easy to decide; the complexity of solving Presburger formulae is at least 2^{2^n}, so the reachability set equivalence problem for Petri nets with semilinear reachability sets is decidable but intractable. Another class of Petri nets for which reachability set equivalence is decidable but intractable is *bounded* Petri nets, in which the number of tokens in every reachable marking is less than some integer bound b. This is the first non-contrived problem whose complexity is non-primitive recursive ([CARD76]).

Part of the reason that the complexity of reachability set equivalence for bounded Petri nets is so high is that it is given in terms of the number of places and transitions in the net rather than the bound b—it is possible to construct a net with $o(n)$ places which is bounded by 2^{2^n}. One way to reduce the complexity is to express it in terms of the bound b. For example, a *safe* net is a net in which no place can hold more than one token (i.e., a net with a bound of 1). For a safe net with n places, there are 2^n possible markings; thus the simple algorithm of enumerating and comparing the elements of the reachability sets could decide whether two safe Petri nets had equal reachability sets in $o(2^n)$ time. In general, the complexity of the equivalence problem for a net in which each of its n places has a bound of b is $o(b^n)$. While this complexity is better, it is still high. It is also the worst case complexity; the average complexity may be much better.

Another way to reduce the complexity of reachability equivalence for bounded Petri nets is to divide the places in the net into k sets such that at most one place in each set is marked at any time. Many safe Petri nets used in practical applications can be divided into such sets in a natural way. If the sets were truly independent of each other, the size of the reachability set would be the product of the number of markings possible in each set. To get an upper bound, we can assume that the places are distributed evenly among the sets, and that any or none of the places in the set can have a token (this results in the largest possible value of the product), giving an upper bound of $(1+n/k)^k$ markings in the reachability set. This complexity has its largest value when each set contains one place ($k = n$); this is possible for any safe Petri net and results in the $o(2^n)$ complexity described previously. However, k can be much smaller for many Petri nets. The minimum upper bound occurs when k is as small as possible; if at most one of all of the places in a Petri net will ever be marked, then $k = n$, and the upper bound is $n + 1$.

The complexity can be reduced further for safe Petri nets which satisfy one further requirement: Exactly one of the places in each of the k sets will be marked at any given time. This is not possible for safe Petri nets in general, but if it can be done for a given net, we can reduce the upper bound to $(n/k)^k$. Note that if $k > n/2$, then either some of the sets will contain only one place (which by definition will always remain marked, and is therefore useless), or some places appear in more than one set. Sharing is permissible as long as the constraints hold, since this only reduces the independence between sets, reducing the size of the reachability set even further. This upper bound has its maximum value when $k = n/e$ (where $e = 2.718\ldots$), or since k must be an integer, when $k = n/3$. The minimum value, as before, occurs when all of the places in the net are in one set, and $k = 1$.

This upper bound has a number of practical implications for using Petri nets to model systems. The number k generally corresponds to the number of separate processes or entities being modeled. For example, a model of a communications protocol might consist of a sender and a receiver, each of which is a subnet which always contains a token, so k would be two in this case. In a Petri net model of a multi-tasking program, k would be equal to the number of tasks in the program, which is generally small. So even though the upper bound is exponential, it is often small enough to be practical. The instances when k is large correspond in general to very complex systems (such as an entire communications network, with hundreds

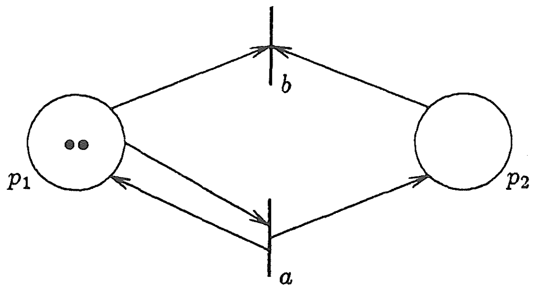

Figure 8
An unbounded Petri net with a regular language

of senders and receivers) which are difficult to analyze as a whole using any kind of model. In these cases, it is often possible to verify the system using induction: verify a simple subset, with a low value of k, then show that an extended system, with k increased by 1, is equivalent to the simple system.

Language Equivalence

Not as much research has been done on Petri net language equivalence as on reachability set equivalence. For general Petri nets, Hack has shown that the equality problem is undecidable for $\mathcal{L}$ and $\mathcal{L}^\lambda$, but that equality for $\mathcal{L}^f$ is reducible to the Petri net reachability problem ([HACK76A]). The reachability problem was later shown to be decidable, but intractable ([MAYR81]). A stronger result was proved by Valk and Vidal-Naquet ([VALK81]): the equivalence problem for two languages in $\mathcal{L}$, with one Petri net having four and the other five unbounded places is undecidable. Analysis of the complexity for decision of language equivalence is harder than that for reachability set equivalence since a Petri net with a finite reachability set may have an infinite language.

One class of Petri nets for which language equivalence is decidable is the class of Petri nets that generate regular languages. This class includes the class of bounded Petri nets, since the reachability graph of a bounded Petri net is isomorphic to a finite automaton. However, unbounded Petri nets exist which also have regular languages, such as the one in Figure 8. It is decidable whether a Petri net has a regular language ([VALK81]). Of course, not every regular language is in $\mathcal{L}$ or $\mathcal{L}^\lambda$ since the latter have the property that all prefixes of strings in the language are also in the language.

For Petri nets with regular languages, a convenient closed form for representing the language is the regular expression. Hunt, Rosenkrantz, and Szymanski have shown several complexity results for the equivalence problem of regular expressions ([HUNT76]). For general regular expressions, and for regular expressions with a star-height greater than or equal to 1, the equivalence problem is CSL-complete. If the star-height is zero (no stars), or the alphabet has only one symbol, the equivalence problem is NP-complete. The equivalence problem is also NP-complete if one or both of the languages being compared is bounded (there exist strings $w_1, w_2, \ldots, w_m$ from Σ^* such that the language $L \subseteq w_1^* w_2^* \cdots w_m^*$, not to be confused with a bounded Petri net). The correlations between the structural properties of a Petri net and a regular expression with equivalent language are not known. Neither is the relationship between the size of a Petri net and the size of a regular expression with an equivalent language. This is an important question—if the regular expression represents the language in a much more compact way than a Petri net, the Petri net language equivalence problem may be tractable.

Another class of Petri net languages which has been studied is the class of *terminal* languages, $\mathcal{L}_0$ (and the corresponding $\mathcal{L}_0^f$ and $\mathcal{L}_0^\lambda$). These languages differ form the $\mathcal{L}$ languages by the additional requirement that the final marking of the net be a specified *terminal* marking in order for the generated string to be in the language. Hack proved that the equivalence problem for $\mathcal{L}_0$ and $\mathcal{L}_0^\lambda$ is undecidable in general ([HACK76A]). A subclass for which equivalence is decidable is the set of regular $\mathcal{L}_0$ languages. The $\mathcal{L}_0$ language generated by a bounded Petri net is regular, but unlike the $\mathcal{L}$ languages, it is undecidable whether the $\mathcal{L}_0$ language generated by an arbitrary Petri net is regular or not. All regular languages can be generated by a $\mathcal{L}_0^\lambda$ type Petri net.

Two other classes of Petri net languages have been described in [PETE81]: the *G-type* and *T-type* languages. The G-type languages are similar to the $\mathcal{L}_0$ languages in that a terminal marking is specified, but the final marking of the net need only cover (i.e., have at least as many tokens in each place as) the terminal marking. The T-type languages require that no transitions be enabled in the final net marking for the string to be in the language (i.e., this is the set of transition firing sequences that lead to a deadlock). Both of these languages have been shown to be subclasses of the $\mathcal{L}_0$ languages, but these classes are largely unexplored. It is not known whether equivalence is decidable for them or not except for the T-type languages with λ-transitions, which are equivalent to the $\mathcal{L}_0^\lambda$ languages.

Other Petri net languages which deserve further study are the languages of infinite words (often referred to as *ω-languages*). These languages reflect the infinite behavior of the Petri net, and could be useful for studying such important properties as livelock-freeness. A Petri net is livelock-free if each transition which is live (potentially firable from any marking) will always eventually fire (i.e., every infinite word in the language includes every transition infinitely often). Valk has initiated work in this area ([VALK83]), but it is mostly exploring the hierarchy of infinite Petri net languages and relating it to other classes of ω-languages. There are still many open questions, such as equivalence issues. Equivalence is almost certainly undecidable in the general case for these languages, but what about simpler cases, such as bounded or safe nets?

Petri Net Extensions

Many extensions have been proposed to make Petri nets more useful. Some of these extensions increase the power of Petri nets; others do not. The most common extension is to allow a place to be tested for zero tokens by allowing *inhibitor arcs*—arcs which disable a transition unless the place is empty. This extension increases the power of Petri nets to that of a Turing machine. Another useful extension is to augment each transition with timing information. Several methods have been proposed—the most general is described in [MERL74]. Merlin added time to Petri nets by associating a minimum and maximum time to each transition; once enabled, the transition can not fire until the minimum time has elapsed, but must fire before the maximum time does. It is easy to show that this extension also gives Petri nets the power of a Turing machine. Another extension that raises the power of a Petri net to a Turing machine is priorities—indicating which transition should be allowed to fire when a conflict exists ([HACK76B]).

One extension that does not increase the power of a Petri net is the addition of color to the tokens ([JENS81]). Transitions in these nets behave differently depending on the color of the tokens on the input places. However, the behavior is essentially the same as if the places and transitions were replicated—once for each color. The main effect of the extension is to reduce the size of the net. Unfortunately, it doesn't reduce the complexity of analysis in general. It should be emphasized that only a finite number of colors is allowed—using an infinite number of colors allows a Turing machine to be simulated. An extension related to colored tokens is the *predicate-transition net* ([GENR81]). This extension associates a predicate with each transition rather than a color, but the idea and usefulness is the same.

Of course, the equivalence problem is undecidable for any extensions that increases or maintains the power of Petri nets. However, combining some of these extensions with the restricted nets described in this paper might be useful. For example, any bounded Petri net, even if powerful extensions are allowed, has the theoretical power of a finite state automaton; the equivalence problem is thus decidable in this case.

Of particular interest is the addition of time, since many Petri nets are designed to model objects that involve time, such as communications protocols and computer processors. One major motivation for adding timing information to Petri nets is to allow the performance of a Petri net modeled system to be analyzed. Most of the Petri nets used with timing are bounded and often safe; the equivalence problem as discussed in this paper is thus decidable for them. However, the definition given here does not compare the timing information of the nets—two nets could be equivalent according to our definition, but one net could perform significantly faster than the other. This is certainly a useful definition for equivalence, but for some applications it may be useful to define two nets to be equivalent only if they take the same amount of time.

Conclusions and Further Research

We have discussed some different definitions of Petri net equivalence, and the complexity of solving the equivalence problem, both of Petri net reachability sets and of Petri net languages. While it is undecidable in general, for common nets it is decidable and often feasible. There are still many areas in which research needs to be done, particularly in the area of Petri net languages and Petri net extensions—especially timed Petri nets.

The definition of reachability sets given in this paper is new—in particular the method of combining and hiding places for reachability set analysis. The impact of this definition on the reachability graph of a Petri net hasn't been explored at all, especially for the $\mathcal{R}$ and $\mathcal{R}^\lambda$ classes. A logical way to generate the reachability graph for these would be to generate the reachability graph the normal way (using the places of the net rather than a transformation on them) and then apply the desired homomorphism to get a new, but similar, reachability graph. This graph, however, would probably have several distinct nodes with indistinct labels—indicating the states that appear the same from the outside, but are different internally. It may be desirable to combine these nodes together to form a *user's view* of the net. For example, consider a Petri net that models a postage stamp machine. The customer puts two dimes into the machine and a postage stamp comes out. The standard reachability graph of this machine would have three states: one representing the initial state of the machine (no money inserted), one for the state where just one dime has been inserted,and one for the state where two dimes have been inserted and a stamp has been ejected. Removing the stamp puts the machine back in the initial state. However, there is really no way to tell the first two states apart without looking inside the machine (except to insert a dime and see what happens). It is possible that a customer may insert one dime in the machine and receive a stamp for it (if someone else previously put just one dime in and left), thus to the customer, the machine has only two states: a stamp is present or one is not. When a stamp is present, removing it puts the machine in the other state. When no stamp is present, inserting a dime will either keep the machine in the same state (apparently doing nothing) or it will eject a stamp. It is hard to know which view is more desirable; the latter needs to be studied. It may also be possible to generate the user's view reachability graph without generating all the states in the complete version (which may be useful if the complete version has hidden unbounded states).

REFERENCES

[ARAK77] Araki, T. and T. Kasami, "Decidable Problems on the Strong Connectivity of Petri Net Reachability Sets," *Theoretical Computer Science*, vol. 4, pp. 97–119, 1977.

[CARD76] Cardoza, E. W., R. J. Lipton, and A. Meyer, "Exponential Space Complete Problems for Petri Nets and Commutative Semigroups," *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, New York: ACM, 1976, pp. 50–54.

[CCIT80] Comité Consultatif Internationale do Télégraphique et Téléphonique, "Draft Revised CCITT Recommendation X.25," *Computer Communication Review*, pp. 56–129, 1982.

[GENR81] Genrich H. J. and K. Lautenbach, "System Modelling with High-Level Petri Nets," *Theoretical Computer Science*, vol. 13, pp. 109–136, 1981.

[GINS66] Ginsburg, S. and E. H. Spanier, "Semigroups, Presburger Formulas, and Languages," *Pacific Journal of Mathematics*, vol. 16, pp. 285–296, 1966.

[GRAB80] Grabowski, J., "The Decidability of Persistance for Vector Addition Systems," *Information Processing Letters*, vol. 11, pp. 20–23, 1980.

[HACK76A] Hack, Michel, "Petri Net Languages," Technical Report 159, Cambridge, Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, 1976.

[HACK76B] Hack, Michel, "Decidability Questions for Petri Nets," Technical Report 161, Cambridge, Massachusetts: Laboratory for Computer Science, Massachusets Institute of Technology, 1976.

[HOPC79] Hopcroft, J. and J. J. Pansiot, "On the Reachability Problem for Five-Dimensional Vector Addition Systems," *Theoretical Computer Science*, vol. 8, pp. 135–159, 1979.

[HUNT76] Hunt, Harry B. III, Daniel J. Rosenkrantz, and Thomas G. Szymanski, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages," *Journal of Computer and System Sciences*, vol. 12, pp. 222–268, 1976.

[JENS81] Jensen, Kurt, "Coloured Petri Nets and the Invariant-Method," *Theoretical Computer Science*, vol. 14, pp. 317–336, 1981.

[MERL74] Merlin, P., "A Study of the Recoverability of Computing Systems," Ph.D. Dissertation, University of California, Irvine: Department of Information and Computer Science, 1974.

[MAYR81] Mayr, Ernst W., "An Algorithm for the General Petri Net Reachability Problem," *Symposium on Theory of Computing*, (Milwaukee 1981), pp. 238–246.

[OPPE78] Oppen, D. C., "A $2^{2^{2^{pn}}}$ Upper Bound on the Complexity of Presburger Arithemetic," *ICSS*, vol. 16, pp. 323-332, 1978.

[PETE81] Peterson, James L., *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs: Prentice-Hall, 1981.

[VALK81] Valk, Rüdiger and Guy Vidal-Naquet, "Petri Nets and Regular Languages," *Journal of Computer and System Sciences*, vol. 23, pp. 299–325, 1981.

[VALK83] Valk, Rüdiger., "Infinite Behaviour of Petri Nets," *Theoretical Computer Science*, vol. 25, pp. 311–341, 1983.

[YAMA81] Yamasaki, Hideki, "On Weak Persistency of Petri Nets," *Information Processing Letters*, vol. 13, pp. 94–97, 1981.

[YAMA84] Yamasaki, Hideki, "Normal Petri Nets," *Theoretical Computer Science*, vol. 31, pp. 307–315, 1984.