Fast Graph Generation for Synchronous Processes

Chris Tofts
Information Infrastructure Laboratory
HP Laboratories Bristol
HPL-2002-16
January 29th, 2002*

E-mail: chris_tofts@hp.com

The automated analysis of concurrent systems requires that they are translated from a syntactic presentation into some semantic space. One common space of interpretation is transition systems (Milner, 1980; Milner, 1990). In particular situations it is possible to produce these transformations piecewise, but in general we need to produce the complete system graph in order to analyse it. Performing this task efficiently is clearly important for the practicality of automated systems analysis. We present some techniques that greatly reduce the cost of generating graphs from synchronous presentations of concurrent systems. These methods have been implemented within an analysis tool the probabilistic workbench (PRWB) (Tofts, 1995).

* Internal Accession Date Only Approved for External Publication

Fast Graph Generation for Synchronous Processes

C. Tofts
HP Research Laboratories Bristol,
Filton Road, Stoke Gifford,
Bristol, BS34 8QZ,
chris_tofts@hp.com

January 29, 2002

Abstract

The automated analysis of concurrent systems requires that they are translated from a syntactic presentation into some semantic space. One common space of interpretation is transition systems[2, 4]. In particular situations it is possible to produce these transformations piecewise, but in general we need to produce the complete system graph in order to analyse it. Performing this task efficiently is clearly important for the practicality of automated systems analysis. We present some techniques that greatly reduce the cost of generating graphs from synchronous presentations of concurrent systems. These methods have been implemented within an analysis tool the probabilistic workbench (PRWB)[7].

1 Introduction

The automation of the analysis of concurrent systems is an important activity. Whilst we have many well founded presentations of these systems, they all inherently lead to a large number of states, which limit the scope of unassisted human analysis. Equally, the scale of these systems leaves hand analysis very prone to error. At the core of such tools [1] is the problem of graph building. In the main the systems are understood as transition graphs, and these are derived from the concurrent system presentation. Given a system with N components each of which can engage in potentially K behaviours at some point then we shall have to consider K^N potential choices if we allow the changes to occur synchronously[1]. Further we still have the problem that the state space is also exponential in the number of parallel components. Clearly if we are to construct the graph representations of large systems we need to pay some attention to the manner in which we compute over our formal systems.

We present a prototypical synchronous process calculus, in this instance with priority and probability extension, but our approach would work for any synchronous system. We then present several approaches to reduce the computational cost of graph generation.

2 WSCCS

The language WSCCS [5, 6] is an extension of Milner's SCCS [3] a language for describing synchronous concurrent systems. Here we provide an introduction to the syntactic constructs that

[1] This makes the comparison of synchronous and asynchronous approaches difficult. An asynchronous system has NK choices but each one results in only one (or sometimes two) state changes. Here we have more choices, but the change in state is greater

underlie WSCCS but omit the formal semantics and algebraic properties as they have a full description elsewhere [6]. For asynchronous probabilistic calculi one may use the methods of [3, 4] to reduce the problem to a synchronous one and our results would then be applicable. The only restriction is that in the later development of this work priority is necessary to model progress. Since WSCCS has all of the requisite descriptive power [6] it is appropriate as a base description language for this problem class.

To define the language we presuppose a free abelian group *Act* over a set of atomic action symbols with identity $\surd$ and the inverse of a being $\overline{a}$. As in SCCS, the complementary actions a (conventionally input) and $\overline{a}$ (output) form the basis of communication. Within our group we define that $\overline{a} = a^{-1}$. For clarity when the action names may not be easily separated we shall denote the multiplication in the free group by the symbol #.

2.1 Expressions

We define a set of expressions.

Definition 2.1 *A* relative frequency expression (RFE) *is formed from the following syntax, with x ranging over a set of variable names VRF, and c ranging over a fixed field (such as $\mathcal{N}$ or $\mathcal{R}$):*

$$e ::= x|c|e + e|e * e$$

Further we assume that the following equations hold for relative frequency expressions:

$$\begin{array}{lcl} e + f & = & f + e \\ (e + f) + g & = & e + (f + g) \\ e * f & = & f * e \\ (e * f) * g & = & e * (f * g) \\ e * (f + g) & = & e * f + e * g \end{array}$$

Alternatively, we have commutative and associative addition and multiplication, with multiplication distributing over addition. We shall assume that two expressions are equivalent if they can be shown so by the above equations.

In the sequel we shall omit the $*$ in expressions, denoting expression multiplication by juxtaposition. It should be noted that unlike other calculi with expressions [4] the value of our expressions can have **no effect** on the structure of the transition graph of our system. Hence we should not expect that adding this extra structure to our probabilistic process algebra will cause any new technical difficulties.

2.2 Weights

We also take a set of weights $\mathcal{W}$, denoted by w_i, which are of the form[2] $e\omega^k$ with e from the relative frequency expressions and the ω^k (with $k \geq 0$) a set of infinite objects, with the following multiplication and addition rules (assuming $k \geq k'$), we consider the objects e used as weights to be abbreviations for $e\omega^0$:

$$e\omega^k + f\omega^{k'} = e\omega^k = f\omega^{k'} + e\omega^k \qquad e\omega^k + f\omega^k = (e + f)\omega^k = f\omega^k + e\omega^k$$
$$e\omega^k * f\omega^{k'} = (ef)\omega^{k+k'} = f\omega^{k'} * e\omega^k$$

[2]Here e is the relative frequency with which this choice should be taken and k is the priority level of this choice. The choice of notation is based in [Tof90] arising from the observation that priority is similar to infinite weight.

2.3 The Calculus

The collection of WSCCS expressions ranged over by E is defined by the following BNF expression, where $a \in Act$, $X \in Var$, $w_i \in \mathcal{W}$, S ranging over renaming functions, those $S : Act \longrightarrow Act$ such that $S(\surd) = \surd$ and $\overline{S(a)} = S(\overline{a})$, action sets $A \subseteq Act$, with $\surd \in A$, and arbitrary *finite* indexing sets I:

$$E ::= X \mid a.E \mid \sum\{w_i E_i | i \in I\} \mid E \times E \mid E\lceil A \mid \Theta(E) \mid E[S] \mid \mu_i \tilde{x}\tilde{E}.$$

We let Pr denote the set of closed expressions; and add $\mathbf{0}$ which is defined by $\mathbf{0} \stackrel{def}{=} \sum\{w_i E_i | i \in \emptyset\}$ to our syntax.

The informal interpretation of our operators is as follows:

- $\mathbf{0}$ a process which cannot proceed;
- X the process bound to the variable X;
- $a : E$ a process which can perform the action a whereby becoming the process described by E;
- $\sum\{w_i.E_i | i \in I\}$ the *weighted* choice between the processes E_i, the weight of the outcome E_i being determined by w_i. We think in terms of repeated experiments on this process and we expect to see over a large number of experiments the process E_i being chosen with a relative frequency of $\frac{w_i}{\Sigma_{i \in I} w_i}$.
- $E \times F$ the synchronous parallel composition of the two processes E and F. At each step each process must perform an action, the composition performing the composition (in Act) of the individual actions;
- $E\lceil A$ represents a process where we only permit actions in the set A. This operator is used to enforce communication and bound the scope of actions;
- $\Theta(E)$ represents taking the prioritised parts of the process E only;
- $E[S]$ represents the process E relabelled by the function S;
- $\mu_i \tilde{x}\tilde{E}$ represents the solution x_i taken from solutions to the mutually recursive equations $\tilde{x} = \tilde{E}$.

Often we shall use a binary plus instead of the two (or more) element indexed sum, thus writing $\sum\{1_1.a : P,\ \ 2_2.b : Q | i \in \{1, 2\}\}$ as $1.a : P \ + \ 2.b : Q$. We allow ourselves to specify processes definitionally, by providing recursive definitions of processes. For example, we write $A \stackrel{def}{=} a.A$ rather than $\mu x.ax$. The weight n is an abbreviation for the weight $n\omega^0$, and the weight w^k is an abbreviation for the weight $1\omega^k$.

For a full description of the operational semantics equivalences and the algebra of WSCCS see [5, 6].

3 Graph Generation

The graph generation problem for a WSCCS system can best be thought of in the following normal form[3]:

[3]Strictly we have two forms with and without the priority operator, but this has no effect on the following presentation.

$$\Theta(P_1 \times P_2 \times \ldots \times P_n)\lceil\{P\}$$

with each of the P_i of the form $\sum_{j=1}^{j=k_i} w_{(i,j)}.a_{(i,j)} : P'_{(i,j)}$. The graph generation problem is then that of studying the products:

$$\Theta(w_{(1,a_1)}w_{(2,a_2)}\ldots w_{(n,a_n)}.a_{(1,a_1)}a_{(2,a_2)}\ldots a_{(n,a_n)} : (P'_{(1,a_1)} \times P'_{(2,a_2)} \times \ldots \times P'_{(n,a_n)})\lceil\{P\}$$

with $a_1 \in [1, k_1]$, $a_2 \in [1, k_2]$, ldots $a_n \in [1, k_n]$. The obvious naive algorithm for computing this product would be to code up[4] the loops as follows (replacing subscripts by array references):

```
for i[1]=1 to k[1]
 for i[2]=1 to k[2]
  ...
   for i[n]=1 to k[n]
    weight=W1[i[1]]*W2[i[2]]*...*Wn[i[n]]
    action=A1[i[1]]*A2[i[2]]*...*An[i[n]]
    process=P1[i[1]]*P2[i[2]]*...*Pn[i[n]]
```

obviously the appropraite multiplication operators are used in each of the products above.

3.1 Unique ID

We assign a unique identifier to each sequential state within the process. Consequently we can order parallel products by using any order on these identifiers. This allows us to resolve syntactic equivalence at the level of associativity and commutativity on the parallel processes.

3.2 Partial Multiplication

In the naive presentation it should be noted that in the final multiplication

```
  ...
   for i[n]=1 to k[n]
    weight=W1[i[1]]*W2[i[2]]*...*Wn[i[n]];
    action=A1[i[1]]*A2[i[2]]*...*An[i[n]];
    process=P1[i[1]]*P2[i[2]]*...*Pn[i[n]];
```

the sections `W1[i[1]]*W2[i[2]]*...*Wn[i[n-1]]`, `A1[i[1]]*A2[i[2]]*...*An[i[n-1]]` and `P1[i[1]]*P2[i[2]]*...*Pn[i[n-1]]` are all fixed for the duration of the loop and therefore should only be evaluated once. Leading to the following implementation:

```
for i[1]=1 to k[1]
{PW[1]=W1[i[1]];
 PA[1]=A1[i[1]];
 PP[1]=P1[i[1]];
 for i[2]=1 to k[2]
 { PW[2]=W2[i[2]]*PW[1];
   PA[2]=A2[i[2]]*PA[1];
   PP[1]=P2[i[2]]*PP[1];
   ...
```

[4] As the number of iterates and their ranges change this presentation is not how the program will actually work but serves to illustrate the alogrithms.

```
   for i[n]=1 to k[n]
   {weight=Wn[i[n]]*PW[n-1];
    action=An[i[n]]*PA[n-1];
    process=Pn[i[n]]*PP[n-1];
   }
 }
}
```

In this version the partial multiplications are kept during the computation and consequently we need only perform one multiplication for each new state rather than $n-1$. Whilst this may seem obvious in this presentation when the ranges of the loops are implicitly coded and the number of loops required are variable, as is the case within an actual implementation, then this calculation method is far from trivial.

3.3 Fan Out Order

The above assumes that the cost of multiplication within the free group on actions is symmetric. It is not. Multiplication on a free group is essentially an insertion problem, and it is much cheaper to insert a short word into a complicated one. Consequently we wish to maximise the number of short insertions. Since the actions on individual processes tend to be simple this can be achieved be handling the processes in inverse fan out order. That is to say if we have processes that have choice numbers 4,2,1,5,3,3 then we exploit the associativity and commutativity of processes to oder the muliplication as 1,2,3,3,4,5. This minimizes the number of complex free group multiplications that our algorithm has to undertake, for the cost of a sort of size n at the start of the main loop described above.

3.4 Repetition

The usual application of automated analysis tools is to largely heterogeneous systems and consequently graph builders do not take advantage of repetition. In our case we are interested in performance and probability. Consequently we often have repetition in our systems and this can be exploited in two ways. Our use of unique identifiers allows us to detect when we have a 'run' of identical processes within a parallel product.

Since processes obey associativity and commutativity then we can exploit the fast positive integer power algorithm for numbers:

$$a^n = \begin{cases} (a^{\lfloor \frac{n}{2} \rfloor})^2 & n \; mod \; 2 = 0 \\ (a^{\lfloor \frac{n}{2} \rfloor})^2 * a & n \; mod \; 2 = 1 \end{cases}$$

this makes the cost of evaluating repetitions logarithmic rather than linear in the power.

Further, in this instance we know that there will be points of commonality in the next states of the system. Consider the trivial coin tossing problem:

$$Coin \stackrel{def}{=} 1.head : Coin + 1.tail : Coin$$
$$Sys \stackrel{def}{=} Coin \times Coin$$

In its full expansion Sys is the following process:

$$Sys = 1.head^2 : Sys + 1.head\#tail : Sys + 1.tail\#head : Sys + 1.tail^2 : Sys$$

but the free product on actions admits associativity and commutativity so the above is equal to

$$Sys = 1.head^2 : Sys + 2.head\#tail : Sys + 1.tail^2 : Sys$$

We can identify this situation by exploiting the unique identifiers, but the cost of the absorbtion requires us to sort the various action and destination state pairs. Performing this operation on all expansions is wasteful as it is unlikely to reduce the number of choices. However, when we have repeated processes it is clear that choices can be reduced by performing absorbtion as above. The reduction of choices is very important in the wider state multiplication algorithm above as it is the dominant cost. For instance for n coins with no absorbtion we have 2^n choices, but actually there are $n + 1$ distinct choices.

4 Conclusions

By exploiting the normal form of processes we can greatly reduce the cost of graph building. In particular in the generation of synchronous actions, where the multitipliation costs within a free group will be high, care needs to be taken with the core generation of the expansion of the processes.

Furthermore in settings where we have repetition, common in performance problems, then we can exploit that repetition to greatly reduce the cost of our expansion. All of the above strategies have been used to improve the core graph generation within the PRWB[7], but have not been documented beyond the source code previously.

References

[1] R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: A Semantics-Based Tool for the Verification of Concurrent Systems. *ACM Transactions on Programming Languages and Systems*, 15(1), 1993.

[2] R. Milner, Calculus of Communicating System, LNCS92, 1980.

[3] R. Milner, Calculi for Synchrony and Asynchrony, Theoretical Computer Science 25(3), pp 267-310, 1983.

[4] R. Milner, Communication and Concurrency, Prentice Hall, 1990.

[5] C. Tofts, A Synchronous Calculus of Relative Frequency, CONCUR '90, Springer Verlag, LNCS 458.

[6] C. Tofts, Processes with Probabilities, Priorities and Time, FACS 6(5): 536-564, 1994.

[7] C. Tofts, Analytic and locally approximate solutions to properties of probabilistic processes, Proceedings TACAS '95, LNCS 1019, 1995.