Efficiently Modelling Resource in a Process Algebra

Chris Tofts
HP Laboratories Bristol
HPL-2003-181
September 2nd , 2003*

E-mail: chris_tofts@hp.com

process algebra, semaphore, resource, composition

In a concurrent system the effects of contention for resource are primary for both understanding and controlling the behaviour of the system. Since resources are inherently shared, hoping for a compositional presentation seems highly unlikely. Equally, in this context, composition in terms of the ability to subdivide resource seems an ambitious goal. In this presentation we demonstrate that, by exploiting synchrony, we can present resources in a divisible manner. Further, this eradicates counting duplication (counts residing both in the resource representation and in the claiming entities) greatly reducing the state space of the system. Finally our compositional representation of resource usage in a synchronous process algebra is obtained without any changes or additions to the underlying language and could be achieved in 'bare' SCCS. For brevity we assume a basic familiarity with asynchronous (such as CCS) and synchronous process algebra (such as SCCS) for an introduction see 'Communication and Concurrency' by Robin Milner, and the SCCS probabilistic/prioritized extension WSCCS.

* Internal Accession Date Only
Approved for External Publication

Efficiently Modelling Resource in a Process Algebra.

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

August 27, 2003

Abstract

In a concurrent system the effects of contention for resource are primary for both understanding and controlling the behaviour of the system. Since resources are inherently shared hoping for a compositional presentation seems highly unlikely. Equally, in this context, composition in terms of the ability to subdivide resource seems an ambitious goal. In this presentation we demonstrate that, by exploiting synchrony, we can present resources in a divisible manner. Further, this eradicates counting duplication (counts residing both in the resource representation and in the claiming entities) greatly reducing the state space of the system. Finally our compositional representation of resource usage in a synchronous process algebra is obtained without any changes or additions to the underlying language and could be achieved in 'bare' SCCS [7]. For brevity we assume a basic familiarity with asynchronous (such as CCS) and synchronous process algebra (such as SCCS) for an introduction see [8] and the SCCS probabilistic/prioritized extension WSCCS[11].

1 Introduction

Most components within concurrent systems can be roughly partitioned into two types:

1. the interesting bits - the components that control interaction and cause computation to proceed;
2. the resource bits - variables, semaphores, counters that record state, both local and global, and are utilised by the interesting components.

In our work on the semantics[5] of DEMOS [1, 2] one of the interesting observations is that the state costs of the 'interesting' parts of the model are small, but the state cost of the 'boring' bits is the main limitation is performing automated formal analysis.

2 Representing resource

Classically resources in process algebras are modelled as semaphores. As an example the simple mutual exclusion problem in CCS[8] is usually presented thus:

$$Sem \stackrel{def}{=} get.put.Sem$$
$$User \stackrel{def}{=} \overline{get}.critical.\overline{put}.User$$
$$ME - Sys \stackrel{def}{=} (User|User|\ldots|User|Sem)\backslash\{get, put\}$$

Off greater interest is the situation when the resource comes in chunks, and a component may want to use less than the totality. From the compositional view the repesentation we should desire would be as follows (exploiting the definition of Sem given above):

$$ResN \stackrel{def}{=} Sem|Sem|\ldots|Sem \ (N \text{ copies})$$
$$User2 \stackrel{def}{=} \overline{get}.\overline{get}.using2.\overline{put}.\overline{put}.User2$$
$$Use2 - Sys = (User2|User2|\ldots|User2|ResN)\backslash\{get, put\}$$

However, elementary analysis (consider the case when $N = 2$), shows that the above representation **does not** behave in the manner we would wish. The fundamental problem is that there is no composition on actions within an asynchronous calculus.

Within the setting of a synchronous calculus such as SCCS [7] or MEIJE[9] we would standardedly proceed in much the same way:

$$SSem \stackrel{def}{=} \$get : \$put : SSem$$
$$User \stackrel{def}{=} \overline{get} : critical : \overline{put} : User$$
$$ME - Sys \stackrel{def}{=} (User|User|\ldots|User|Sem)\lceil\{critical\}$$

in the above \$ denotes a desynchronised or waiting capable action.

However, with multiple resource use we would write the previous example as follows:

$$ResN \stackrel{def}{=} SSem|SSem|\ldots|SSem \ (N \text{ copies})$$
$$User2 \stackrel{def}{=} \overline{get^2} : \$using2 : \overline{put^2} : User2$$
$$Use2 - Sys \stackrel{def}{=} (User2|User2|\ldots|User2|ResN)\lceil\{using2\}$$

and this will indeed proceed in the manner in which we would expect. In some senses we have a different way to express a value passing calculus in the synchronous setting. Rather than subdiving actions we exploit the powers. In this instance we can subdivide the resource by exploiting the associativity and commutativity of parallel composition. Then renaming the local access to the divided resource will allow the movement the permission operators leading to the representation of the system as a pure parallel composition.

Whilst the above representation is compositional it duplicates state information, in order to return resources entities must know how much they hold hence the multipe resource need not count them. This has no impact at the level of the complete system but has major implications for the efficiency of automated graph generation systems. Increasing state has multiplicative effect on the number of states that need to be considered. Taking this representational requirement we could define resources as follows:

$$SpSem \stackrel{def}{=} \$get : \$\mathbf{0}$$
$$ResN \stackrel{def}{=} SpSem|SpSem|\ldots|SpSem \ (N \text{ copies})$$
$$User2 \stackrel{def}{=} \overline{get^2} : using2 : (SpSem|SpSem|User2)$$
$$Use2 - Sys \stackrel{def}{=} (User2|User2|\ldots|User2|ResN)\lceil\{using2\}$$

largely this approach is avoided because it uses spawning, at therefore a risk of being infinite state. Equally early graph builders did not absorb the \$**0** objects allowing the syntactic form to grow indefinately.

The above approaches seem to hint that it should be possible to encode all of the resource requirements purely within the action part of the process system which would then minimise states that need to be considered in automated constructions. So we present a further alternative view of resource:

$User2 \stackrel{def}{=} \$using2\#have^2 : User2$

$Use2 - Sys \stackrel{def}{=} (User2|User2|\ldots|User2)\lceil\{\{\surd, using2\}\#\{\surd, have^1, \ldots have^N\}\}$

In this instance we need a bit of notation to form permission sets, the above implies the free group formed by the combination of the atom ($using2$) and the action set. Explicitly we have the permission set:

$$\{\surd, using2, using2\#have^2, using2\#have^4, using2\#have^6, \ldots\}$$

as a consequence we do not permit more than one instance of a $using2$ action to be occuring on any tick. Hence, the permission has the effect of restricting the number of copies of $using2$ and thus implicitly enforces the resource limit. If we changed the permission to

$$\{\{\surd, using2, using2^2, using2^3\}\#\{\surd, have^1, \ldots have^N\}\}$$

this would have the effect of permitting up to 3 instances of the user being live at the same time. With this representation, without **any** extension to the basic language of SCCS, we have gained three major benefits:

1. we have no need for an explicit state representation of the resource(semaphore);
2. we can vary the amount of resource available by changing the permission set, and do not have to change any of the component process descriptions;
3. the actions witnessing the current levels of resource utilisation are **always** visible.

One complaint about the above might be that it will only work when the ownership of a resource extends for precisely one 'tick'. Consider the following process:

$User2 \stackrel{def}{=} \$using2\#have^2 : User2a$

$User2a \stackrel{def}{=} using2\#have^2 : User2b$

$User2b \stackrel{def}{=} using2\#have^2 : User2$

$Use2 - Sys \stackrel{def}{=} (User2|User2|\ldots|User2)\lceil\{using2\#\{have^1, \ldots have^N\}\}$

Note that in the states $Using2a$ and $Using2b$ to proceed we must be able to perform the action $have^2$ there is *no alternative*. If this does not occur the system will deadlock. Hence once ownership is established it will be maintained by forcing the other system components to wait. It should be noted that any individual deadlocked process acts as an anihilator for the complete system within a synchronous calculus. Consequently, any failure to proceed on the part of a process of the form of $User2a$ and $User2b$ will, in essence, destroy the system under study.

Whilst the above has enabled us to remove the explicit resource processes completely some tools do not support limited action specification in this instance we can complete the system for the cost of a single state process as follows:

$Res \stackrel{def}{=} \surd : Res + \overline{have} : Res + \overline{have^2} : Res + \ldots + \overline{have^N} : Res$

$Use2 - Sys \stackrel{def}{=} (User2|User2|\ldots|User2|Res)\lceil\{using2\}$

In the setting of a probabilistic calculus the above approach permits compositional resource usage reasoning. For each particular set of subcomponents we can compute the average demand on resource and then combine these estimates together. Whilst in any interesting case this will require more resource than will be avialable we can obviously estimate the period for which the system resource requirements exceed those that are available and hence the performance limitations imposed by the resource restrictions. Such a calculational approach can be embedded in the toolset presented in [12]. As a large scale example we represent the model of printer memory usage derived by Athena Christodolou in [3]

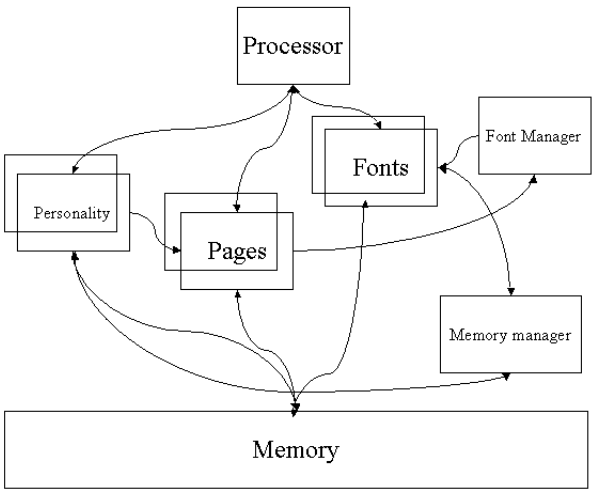

Figure 1: Interactions diagram for printer memory usage.

3 Printer memory usage

In Figure 1 the resource usages within a simplified printer model are presented, this is largely based on the model presented in [3] and requires WSCCS [12] to provide the rate and priority constructions. The various elements within the printer compete for two main resources the processor and memory.

3.1 Personality

Within a printer the personality is the element which converts the language in which the page is presented (e.g., pdf, pgl ,ps) into a pattern of dots which can be placed on the paper by a print engine. A printer may contain several different personalities. A personality has several basic states:

- Active: pages are recieved and then parsed;
- Parsing: using the processor the page is converted from the presentation format to the printing format;
- Idle: the printer is now using a different personality;
- Pickled: in this state the personality returns some of its working memory and compresess its internal state subsequently it has to be reconstructed. Furthermore, it can be killed back to its initial state, whereby it gives up all of its working memory and internal state;
- Initialising: the personality has to construct its internal working tables and gather working memory.

Constructing each of the above states formally.

Idle State

$$PI \stackrel{def}{=} 1.context : PGM \\ +1.\surd : PI$$

If a context switch occurs then this personality becomes the active personality so it needs to acquire some working memory state PGM (personality get memory), otherwise it stays idle.

Getting memory

$$PGM \stackrel{def}{=} 1\omega^2.get^2 : Pinit \\ +1\omega.free : PGM \\ +1.\surd : PGM$$

The personality tries to get the 2 units of memory it needs, if they are not available it asks the memory manager to free some memory and retries, if the memory manager cannot respond then it retries. If it succeeds in acquiring its working memory then it needs to set up its internal state.

Initialising

$$Pinit \stackrel{def}{=} pdone.get^2\#proc : Pactive \\ +(1 - pdone).get^2\#proc : Pinit \\ +1.get^2 : Pinit$$

In this state we have two units of memory so **all** transitions from it must have get^2 actions. We use an exponential distribution parameterised by *pdone* to represent the time taken to construct the internal data tables. But to do that we need to utilise the processor. So we run interleaved for an exponential time on the processor, the two transitions with *proc* as part of the action. Otherwise, it simply maintians its ownership of the memory.

Once it has been initialised it is active and can accept pages.

Active

$$Pactive \stackrel{def}{=} pages.get^2 : Pparse \\ +(1 - pages).get^2\#\overline{context} : Pidle$$

In this state a number of pages arrive at an exponential rate given by the variable *pages*. Note as this state has memory all transitions have get^2 actions. If a page arrives then it is parsed. If there are no more pages for this personality then the context is switched to an alternate one.

Parsing

$$Pparse \stackrel{def}{=} dl.get^2\#page\#proc : (Page|Pactive) \\ +(1 - dl).get^2\#proc\#proc : Pparse \\ +1.get^2 : Pparse$$

Since no system can have an unbounded number of pages there is a global limit on the number of pages simultaneously being processed, the requirement of a *page* action before the parsing can commence. The parsing time is given by the parameter *dl* as an exponential distribution, and we also require the processor in order that we can perform the parse. At this point we can *spawn* a page that will run its own processing throughtout the rest of the printing process. In the spawn the personality is returned to the active state to see if there is another page for it.

Idle

$$\begin{aligned} Pidle \stackrel{def}{=} \; & 1.get^2 : Pidle \\ & +1.pickle\#get^2 : Ppickle \\ & +1.get^2\#context : Pactive \end{aligned}$$

An idle personality still has 2 units of memory so all transitions must have a get^2 action. From the idle state it may get a context switch instruction to become live and will return to the active state. Equally it may be asked to compress itself, 'pickling' which halves its memory requirements. Otherwise nothing may happen, so it retains its two memory units.

Pickling

$$\begin{aligned} Ppickle \stackrel{def}{=} \; & pt.get^2\#proc : Ppickled \\ & +(1-pt).get^2\#proc : Ppickle \\ & +1.get^2 : Ppickle \end{aligned}$$

The pickling process occupies the processor for an amount of time given by an exponential distribution pt, during this 2 units of memory are required. If the processor is not available it just retains its claim on the 2 units of memory.

Pickled

$$\begin{aligned} Ppickled \stackrel{def}{=} \; & 1.get\#context : PG1 \\ & +1.get : Ppickled \\ & +1.get\#killpers : PI \end{aligned}$$

In this state the personality only has 1 unit of memory so all transitions have a get component. If a context switch occurs then the personality must 'unpickle' itself. Alternatively its last unit of memory may be reclaimed and it is 'killed' back to its initial state.

Unpickling

$$\begin{aligned} PG1 \stackrel{def}{=} \; & 1\omega^2.get^2 : Pinit \\ & +1\omega.free\#get : PG1 \\ & +1.get : PG1 \end{aligned}$$

We assume unpickling is fast and does not require significant processing. In this instance we must have get on all transitions to indicate the possesion of 1 unit of memory, but it needs two. So it attempts to claim it. If this claim cannot be met then it asks the memory manager to free some memory for it and tries again. If the memory manager will not respond then it retries.

3.2 Pages

In this model pages come in two varieties:

- Simple: a page occupying one memory unit which uses one font which is already in memory;
- Complex: a page occupying two memory units which uses three fonts of which one will be new.

Page type choice

$$\begin{aligned} Page \stackrel{def}{=} \; & pcomp.page : CompPg \\ & +(1-pcomp).page : SimpPg \end{aligned}$$

With probability *pcomp* it is a complicated page, otherwise a simple page.

Complex get memory

$$\begin{aligned} CompPg \stackrel{def}{=} \; & 1\omega^2.get^2\#page : CompPgI \\ & +1\omega.free\#page : CompPg \\ & +1.page : CompPg \end{aligned}$$

A complicated page needs 2 units of memory so try to claim them, as per usual if they cannot be claimed ask the memoy manager to free some memory and retry until we can progress.

Complex initialise fonts

$$\begin{aligned} CompPgI \stackrel{def}{=} \; & 1\omega^3.\overline{getFont}^2\#\overline{newFont}\#get^2\#page : CompPgWF \\ & +1\omega^2.\overline{getFont}\#\overline{newFont}^2\#get^2\#page : CompPgWF2 \\ & +1\omega.\overline{newFont}^3\#get^2\#page : CompPgWF3 \\ & +1.get^2\#page : CompPgI \end{aligned}$$

This has 2 units of memory and uses one page slot, so on **all** transitions we must have the action component $get^2\#page$ to keep its resources. A complicated page needs 3 fonts, 2 old and 1 new. Obviously if 3 fonts are not avialable the extra fonts must be created. So the transitions represent creating 1,2 or 3 fonts depending on the current number of old fonts around in priority order. After the page has launched the font creation, it must wait for the number of fonts it asked for to be built.

Waiting for fonts

The following states represent the system waiting for 1,2 or 3 fonts to be built respectively.

$$\begin{aligned} CompPgWF \stackrel{def}{=} \; & 1.get^2\#crFont\#page : CompPgR \\ & +1.get^2\#page : CompPgWF \\ CompPgWF2 \stackrel{def}{=} \; & 1.get^2\#crFont^2\#page : CompPgR \\ & +1.get^2\#page : CompPgWF2 \\ CompPgWF3 \stackrel{def}{=} \; & 1.get^2\#crFont^3\#page : CompPgR \\ & +1.get^2\#page : CompPgWF3 \end{aligned}$$

As before we must have $get^2\#page$ as an element of all transitions. We ensure that the font will continue to inform the page that it is built until the page accepts the action. Consequently we can wait for all of our fonts to be *simultaneously* ready before proceeding.

Page running

$$\begin{aligned} CompPgR \stackrel{def}{=} \; & cr.get^2\#\overline{fontRelease}^3\#pgDn\#page\#proc : T \\ & +(1-cr).get^2\#proc\#page : CompPgR + 1.get^2\#page : CompPgR \end{aligned}$$

In this instance we need $get^2\#page$ on all transitions for resource retention. The running time of the page is given by the exponential defined by cr. When the page has completed it releases all of its fonts. It then becomes a process that runs forever but produces no actions other than $\surd$, the identity for multiplication.

Simple page get memory

$$\begin{aligned} SimpPg \stackrel{def}{=} & \ 1\omega^2.get^1\#page : SimpPgI \\ & +1\omega^1.free\#page : SimpPg \\ & +1.page : SimpPg \end{aligned}$$

A simple page needs 1 unit of memory so try to claim it, as per usual if they cannot be claimed ask the memoy manager to free some memory and retry until we can progress.

Simple page get font

$$\begin{aligned} SimpPgI \stackrel{def}{=} & \ 1\omega^2.\overline{getFont}\#get\#page : SimpPgR \\ & 1\omega^1.\overline{newFont}\#get\#page : SimpPgWF \\ & +1.get\#page : SimpPgI \end{aligned}$$

The simple page keeps its memory and page slot with $get\#page$ actions on all transitions. It uses one, preferably old, font. If the font is not avialable it is built and the page must wait for that to complete.

Simple page waiting font

$$\begin{aligned} SimpPgWF \stackrel{def}{=} & \ 1.get\#crFont\#page : SimpPgR \\ & +1.get\#page : SimpPgWF \end{aligned}$$

It preserves its resources and waits for a font built return.

Simple page waiting font

$$\begin{aligned} SimpPgR \stackrel{def}{=} & \ sim.get\#\overline{fontRelease}\#pgDn\#page\#proc : T \\ & +(1-sim).get\#proc\#page : SimpPgR \\ & +1.get\#page : SimpPgR \end{aligned}$$

The running time is given by the exponential distribution defined by sim. When running it needs the processor as well as the page slot and memory unit which it requires all the time. When it has finished running it releases its font.

3.3 Fonts and font manager

The lifecycle of a font passes through the following stages:

- Memory acquisition: the memory in which the font will reside must be obtained;
- Font construction: the processor must be used to fill the memory with the font description;
- Font use: the font is used in the construction of a page;
- Font idle: once used the font is available for further printing until its memory is reclaimed by the memory manager.

The font manager simply spawns the number of fonts required by the current pages.

Font manager

$$\begin{aligned} FntMgr \stackrel{def}{=} \; & 1.newFont : (FntMgr|Font) \\ & +1.newFont^2 : (FntMgr|Font|Font) \\ & +1.newFont^3 : (FntMgr|Font|Font|Font) \\ & +1.\surd : FntMgr \end{aligned}$$

Simply spawns the number of fonts requested if no font is requested then do nothing. This is limited to 3 fonts but could be made wider if necessary.

Font claiming memory

$$\begin{aligned} Font \stackrel{def}{=} \; & 1\omega^2.get\#mkfnt : FontB \\ & +1\omega.free : Font \\ & +1.\surd : Font \end{aligned}$$

Try to claim one unit of memory, if not succesful try to get the memory manager to free some memory. Retry until it gets the memory. The action $mkfnt$ is to allow us to see how often new fonts are created.

Font building

$$\begin{aligned} FontB \stackrel{def}{=} \; & fb.get\#proc : FontBt \\ & +(1-fb).get\#proc : FontB \\ & +1.get : FontB \end{aligned}$$

Taking a time given by an exponential distribution with parameter fb. Use the processor and the font's memory to build a font. If the processor is not avialable then wait. Note we must issue *get* on all transitions to retain our memory.

Font built

$$\begin{aligned} FontBt \stackrel{def}{=} \; & 1\omega 1.get\#\overline{crFont} : FontInUse \\ & +1.get : FontBt \end{aligned}$$

Try to inform the customer that the font they requested is available. Again we must issue *get* on all transitions to witness the ownership of the resource. Clearly this new font must be being used, otherwise it would not have been created.

Font in use

$$\begin{aligned} FontInUse \stackrel{def}{=} \; & 1.get : FontInUse \\ & +1.get\#fontRelease : FontUsed \end{aligned}$$

Witness the use of the memory while it is use. Eventually it will be released by its current page. At this point the font can still reside in memory to save recreation time.

Font idle

$$\begin{aligned} FontUsed \stackrel{def}{=} \; & 1.get\#getFont : FontInUse \\ & +1.freeFont\#get : T \\ & +1.get : FontUsed \end{aligned}$$

In this state the font may return to use with a new page, or it may be told to die by the memory manager to release its memory. If neither of these things happen then it continues to occupy 1 unit of memory.

3.4 Memory Manager.

The memory manager responds to free requests from the various components that require memory in order to proceed. If a memory request is refused then these components immediately respond by requesting that memory is made available for them. In this version the memory manager reclaims memory in the following order:

- Killing off unused fonts;
- Asking personalities to pickle themselves;
- Asking personalities to kill themselves.

Memory manager

$$\begin{array}{rcl} MMI & \stackrel{def}{=} & 1.\overline{free} : MM + 1.\surd : MMI \\ MM & \stackrel{def}{=} & 1\omega^5.\overline{freeFont}\#ff : MMI \\ & & +1\omega^4.\overline{pickle}\#fp : MMI \\ & & +1\omega^3.\overline{killpers}\#fk : MMI \\ & & +1.\surd : MMI \end{array}$$

The memory manager waits for a $free$ request. When it sees it it tries the various potential sources of memory in order. It there is no memory to be freed it waits for the next request.

3.5 Putting it all together

We form the system in two stages, firstly building the personality system PS, secondly the complete system SYS.

$$\begin{array}{rcl} IM & \stackrel{def}{=} & \{c1, c2, pgDn, mkfnt, ff, fk, fp, get, proc, page, freeFont, free, pickle, \\ & & killpers, newFont, getFont, fontRelease, crFont\} \\ PS & \stackrel{def}{=} & \Theta((PI|Pactive)\lceil IM) \\ P & \stackrel{def}{=} & \{\{\surd, proc\}\#\{\surd, page, page^2, page^3\}\#\{\surd, get, get^2, \ldots, get^{10}\}\# \\ & & \{\surd, c1, c2, pgDn, mkfnt, ff, fk, fp\}\} \\ SYS & \stackrel{def}{=} & \Theta((PS|FntMgr|MMI)\lceil P) \end{array}$$

Whilst we only start with four parallel components it should be remembered that the font manager and the personality both spawn processes, fonts and pages respectively. If we set a page limit of three and a memory limit of 10 units then we could have a further further 12 processes in the system giving us a grand total of 16 parallel elements. With these scale settings building the graph upon which we can take measurements takes about 2 hours on a P600 with 512Mb of memory. Whilst this model is based on [3] it is considerably more complex, allowing page variation and more fonts. However, on comparable computation resource, the model construction time is much shorter. The abstraction over resource saves a factor of 88. The state costs of the various resources having a multiplicative effect proc resource being 2, the memory 11 and the page limit 4.

4 Further abstraction

The above approach admits an interesting question. Since we have been able to account for resources seperately can we perform our overall resource usage calculations in a more compositional manner?

In the printer model we could replace the explicit font construction calls with a probability from run averages, and replace the page lifetimes with the appropriate averages, taking account of the font wait periods. Equally font lifetimes could be introduced as probabilities. In other words abstract the activities of the memory and font managers as probabilities and move the responsibility onto the respective components. In this instance we could factor the system into Personalities, Pages and Fonts. We could then calculate the stable distributions for each of these elements seperately and then combine the dervied distributions for memory usage. Obviously this distribution could well exceed our hard memory limit. However, the amount of distribution lying above the limit can be calculated and the aditional stalled time that this represents added to the performance parameters that we are calculating. This approximation approach should be valid so long as the usage model of the resources does not include hysterysis.

5 Conclusions

By representing the use of resources in a different fashion we have been able to greatly reduce the amount of intermediate states needed within a graph construction. Furthermore we save the extra cost imposed by the presence of an explicit process representing the resource. As in our large example if we have resources in a system of size 1, 3 and 10 then the state saving is 2*4*11 or a factor of 88 in the size of the graph that has to be constructed.

Our resource representation actually required **no** changes to be made to the underlying processes calculus. Simply be exploiting the standard operators in a novel way we have achieved a resource represenation that is compositional, allows tracking, is easy to vary within a tool and avoids the need for state duplication within the model.

References

[1] G. Birtwistle, DEMOS — discrete event modelling on Simula. Macmillen, 1979.

[2] G. Birtwistle. The Demos Implementation Guide and Reference Manual. Technical Report, 260 pages, Computer Science Department, University of Calgary, 1983.

[3] A. Christodolou, Formal Solutions to Large Scale Performance Problems, PhD Thesis, University of Leeds, 1999.

[4] 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.

[5] G. Birtwistle and C. Tofts, A Denotational Semantics for Process Orientated Simulation Languages, ACM ToMACS 281 - 305:8(3),1998.

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

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

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

[9] R. de Simone, Higher-level synchronising devices in Meije-SCCS. *Transactions in Computer Science* **37**, 245-267, 1985.

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

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

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

[13] C. Tofts, Symbolic approaches to probability distributions in process algebra, BCS FACS 12:392-415, 2000.