Formal Aspects of Computing (1993) 5: 467–479
© 1993 BCS

Formal Aspects of Computing

A CCS-based Investigation of Deadlock in a Multi-process Electronic Mail System

Gordon Brebner

Department of Computer Science, University of Edinburgh, Edinburgh, UK

Keywords: Multi-process software; Deadlock; CCS; Concurrency workbench

Abstract. The networking software for a VAX/VMS computer system had been implemented as a collection of communicating processes. One night, an unusually high load on the electronic mail component of the software caused deadlock to occur between two of the processes. This paper describes how the deadlock was analysed by modelling the software using the Calculus of Communicating Systems (CCS) and then by investigating the behaviour of the model using the Edinburgh Concurrency Workbench (CWB). The analysis suggested how the software should be restructured to prevent the problem recurring; the new set of processes was analysed, and shown to be deadlock-free.

1. Introduction

The University of Edinburgh has operated packet-switched computer networks connecting its various sites since the early 1970's; these networks have supported remote logins, file transfers and electronic mail. Initially, the networks made use of internally-designed communication protocols but, as standard protocols emerged, the networks evolved to make use of them. With the advent of standard wide-area networks such as the British academic community's JANET and British Telecom's PSS, the Edinburgh networks have become part of a world-wide networking community. This has enabled people and computers to interact in many ways, some of them unforeseen by the original network developers.

The Department of Computer Science at the University of Edinburgh has attached DEC VAX/VMS computer systems to the University networks since

Correspondence and offprint requests to: Gordon Brebner, Department of Computer Science, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road, Edinburgh EH9 3JZ, UK.

1980. The necessary communications software has evolved with the networks, and has all been produced locally, usually predating proprietary offerings for VAX/VMS. In the version referred to here, the software implemented the CCITT X.25 networking protocol, together with a CCITT XXX remote login facility, a JNT 'Blue Book' file transfer facility and a JNT 'Grey Book' electronic mail facility [Tan88, Mar91].

This paper is concerned with a problem that occurred in the software implementation of the network electronic mail facility. The problem arose from the activities of an electronic 'Dr Who Fan Club'. One user of a VAX/VMS system received Dr Who mail messages from a source in America and then, using automatic mail forwarding from his account, sent a copy of each message to around twenty other people on different computer systems throughout Britain. One night, after a period of constipation in an American mail system, around thirty Dr Who mail messages arrived for the user in close succession. As would be expected, this caused a short-term load upon the communications software but, unexpectedly, it led to complete deadlock within the mail handling software, a fact discovered the following morning.

To explain how deadlock was possible, it is necessary to examine briefly the overall structure of the communications software. This had a multi-process implementation, with one VMS process handling each communication protocol used: one dealing with the X.25 protocol and controlling the physical link driver, one dealing with the XXX protocol, one dealing with the 'Blue Book' file transfer protocol and one dealing with the 'Grey Book' electronic mail protocol. These processes interacted with each other, as well as with some system and user processes. Interaction between the X.25 process and the other processes was via semaphored access to a shared memory area (for speed); all other inter-process interaction was by data-passing through VMS inter-process mailboxes used as unidirectional data buffers. To avoid any confusion with the electronic mail application under consideration, inter-process "mailboxes" will henceforth be referred to here as inter-process "buffers".

Inspection of the deadlocked software revealed that the file transfer and mail processes were both stuck, each waiting to send data to the other but neither able to, because the inter-process buffers in both directions were full. After consulting the logs kept by the two processes, an analysis of the events preceding the deadlock gave an indication of what had happened to cause the problem, and suggested a possible restructuring of the software implementation that would prevent deadlock happening again.

To investigate the deadlock more thoroughly, it was decided to model the electronic mail part of the communication software so that its behaviour could be formally analysed. The three main aims were to confirm the explanation of why deadlock had occurred, to check whether other forms of deadlock were possible under extreme loading conditions, and to check that any restructured system was deadlock-free.

In the next section, the Calculus of Communicating Systems (CCS), and its application to deadlock investigation using the Edinburgh Concurrency Workbench (CWB), is introduced. Then, three sections describe the analysis of the original mail system, some experiments on the nature of the deadlock, and the analysis of the revised mail system, respectively. The final section draws some conclusions about the usefulness of automated formal methods.

2. CCS and Deadlock Investigation

The Calculus of Communicating Systems (CCS) [Mil89] provides a semantic basis for reasoning about concurrent and communicating systems. Systems are described in terms of *agents*, which are identified here by names beginning with upper-case letters. Agents perform *actions*, evolving to become new (and usually different) agents after each action. Communication between two agents is possible by one agent performing an *output* action and the other agent performing a complementary *input* action simultaneously. Actions are identified here by names beginning with lower-case letters, with output actions having bars over identifiers and sharing identifiers with their complementary input actions (if any).

Agents are defined using CCS primitives. There is a basic agent **0** that can perform no actions, but it is not required here. The following operators (together with recursive agent definitions) will actually be used:

1. Prefix: if P is an agent, then $a.P$ is an agent that performs action a and then behaves like P;
2. Choice: if P and Q are agents, then $P+Q$ is an agent that behaves either like P or like Q non-deterministically;
3. Parallel composition: if P and Q are agents, then $P \mid Q$ is an agent that has the combined behaviour of both P and Q, with communication possible between P and Q if they can perform complementary output and input actions;
4. Restriction: if P is an agent and a is an action, then $P \setminus a$ is an agent that behaves like P except that it cannot perform actions a or $\bar{a}$ *externally*, although these actions (which are complementary) can still be performed for communication *internally* (as shorthand, a set of restricted actions may appear on the right-hand side of the "\");
5. Relabelling: if P is an agent and a and b are actions, then $P[b/a]$ is an agent that behaves like P except that, if P can perform actions a or $\bar{a}$, the relabelled agent can perform actions b or $\bar{b}$ respectively instead (as shorthand, a list of relabellings may appear within the "[" and "]" brackets).

Examples of the operators in use follow in the next three sections.

CCS possesses an 'invisible' action τ that represents an action performed by agents internally with no externally-visible effect. This action is used here in agent expressions of the form $\tau.P + \tau.Q$, which can be regarded as representing an agent that makes an *internal* non-deterministic choice to evolve either to P or to Q; this is different from just $P+Q$, where external factors may influence the 'non-deterministic' choice.

This work has made extensive use of the Edinburgh Concurrency Workbench (CWB), an automated tool for analysing systems expressed in CCS [Cle90]. Definitions of CCS agents can be input to the CWB, which can then answer certain questions about the behaviour of these agents. For investigating deadlock, which is modelled by a particular agent not being able to perform any action, two facilities of the CWB are particularly useful.

The first is the deadlock-finding facility which, given an agent, examines it and all of its evolutionary descendents to check whether any are deadlocked. While this facility is exactly what is required here, the number of different descendents is typically exponential in the number of parallel-composed sub-agents, meaning that, for reasons of both time and space complexity, only relatively simple agents can be examined. Morley [Mor90] has devised some heuristics for reducing the

size of the searched family tree, which improve the situation. Less subtly, the onset of faster computers with more memory has benefitted the work described here.

The second useful CWB facility is that for model-checking which, given an agent and a specification written in a modal logic based on the propositional mu-calculus [Koz83, Sti89], checks whether the agent meets the specification. In particular, this can be used to check whether an agent meets the specification that it is deadlock-free. One possible specification is the recursive formula (using notation explained below):

$$\nu X.\ ((<.>\ \mathbf{true}) \wedge ([.]X))$$

which states that, at all stages of its evolutionary history, the agent can always perform some action. The $<.>$ **true** term involves the *possibility* operator, and specifies that an agent is able to perform some action and then evolves to an agent that satisfies the tautological specification **true**. The $[.]X$ term involves the *necessity* operator, and specifies that, after an agent has performed any action, it always evolves to an agent that satisfies the specification X. However, X is defined recursively, using the νX maximal fixpoint operator, as the conjunction of these two terms, giving an appropriate specification.

The above specification allows an agent to evolve to a point where it only performs internal τ actions thereafter; thus, as far as an external observer is concerned, it has become deadlocked. The specification can be improved to exclude such cases by replacing the $<.>$ **true** term by the more demanding term

$$\mu Y.\ ((<.-\tau>\ \mathbf{true}) \vee (<.>\ Y))$$

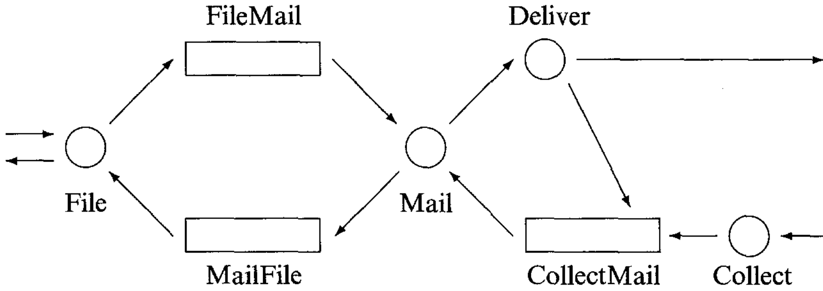
in which Y specifies that an agent either can perform some non-τ action (the $<.-\tau>$ **true** sub-term) or can perform some action and then evolve to an agent that satisfies Y (the $<.>\ Y$ sub-term). This recursive definition of Y uses the μY minimal fixpoint operator, which means that the term specifies that an agent, or one of its descendents, is capable of performing a non-τ action. The revised specification of deadlock-freedom is consistent with that applied by the deadlock-finding facility of the CWB.

The model-checking facility, like the deadlock-finding facility, potentially has an exponential run-time. However, it is sometimes faster here because it only investigates the family tree of an agent as far as necessary to determine whether the agent meets the specification of deadlock-freedom, whereas the deadlock-finding facility checks the whole of the family tree and reports all possible deadlocks. Both facilities have been used in this work, with the slower but more informative facility being more useful. Full details of how the CWB was used are not central to the remainder of the paper but, for the benefit of the interested reader, some further information (including execution times) is included as an appendix.

The CWB is only one of a number of tools that are available to assist with the analysis of concurrent systems; a survey and comparison of many tools can be found in [InP91]. Other tools would also have been suitable for the work described here. Examples include TAV [LGZ89], which allows model-checking using CCS descriptions, and AUTO [BdR90], which allows both deadlock-finding and model-checking using MEIJE [Bou85] descriptions. The particular choice of the CWB as the vehicle for this work was largely due to its convenient local availability.

3. Analysis of the Original Mail System

CCS was used to model the process interactions within the part of the communications software that dealt with electronic mail. Note that this software only handled mail to or from other computers: local mail was dealt with by the standard VAX/VMS mail facilities. The relevant mail system consisted of four processes and three inter-process buffers, inter-connected as shown below:



Circles indicate processes and rectangles indicate buffers; each component is labelled by the identifier of the CCS agent that models it. Arrows indicate data flows, four of which are between the outside world and the mail system: files containing mail are received from, and sent to, networking facilities (arrows at the left-hand side of the figure), and mail messages are collected from, and delivered to, computer users (arrows at the right-hand side of the figure). The significance of the internal arrows will be explained in later paragraphs.

The CCS agent modelling the mail system is capable of performing two types of externally-visible input action: receiving a mail file from the network, and receiving a mail message from a user; and two types of externally-visible output action: sending a file to the network, and sending a mail message to a user. The agent is defined as the parallel composition of four sub-agents corresponding to the four processes, and three sub-agents corresponding to the three inter-process buffers, with restriction being used to hide internal communication actions from external observation. The overall definition is:

```
(Collect
 | CollectMail
 | Deliver
 | Mail
 | MailFile
 | FileMail
 | File)
\{cm_insert, cm_remove, cm_empty,
  md_start, md_stop,
  mf_insert, mf_remove, mf_empty,
  fm_insert, fm_remove, fm_empty}
```

in terms of sub-agents and actions which will now be defined.

First, the agents corresponding to inter-process buffers are all relabelled versions of a standard buffer defined by:

$$\text{Buffer} \stackrel{\text{def}}{=} \text{insert}.\overline{\text{remove}}.\text{Buffer} + \overline{\text{empty}}.\text{Buffer}$$

which can hold only one item: it either can input an item and then output it, or can output a signal that it is empty. The latter capability is necessary to allow other agents to check whether buffers are empty, as will be seen soon. The buffers in the real system had somewhat larger capacities than this, allowing deadlock to be avoided or deferred in most circumstances, but not prevented in extreme circumstances. The more limited buffering in the model gives 'harsher' behaviour with a focus on deadlock potential; this simplification makes the deadlock investigation more tractible. The three actual agents are:

$$\begin{aligned} \text{CollectMail} &\stackrel{\text{def}}{=} \text{Buffer}\,[\text{cm_insert/insert, cm_remove/remove,} \\ &\qquad\qquad \text{cm_empty/empty}] \\ \text{MailFile} &\stackrel{\text{def}}{=} \text{Buffer}\,[\text{mf_insert/insert, mf_remove/remove,} \\ &\qquad\qquad \text{mf_empty/empty}] \\ \text{FileMail} &\stackrel{\text{def}}{=} \text{Buffer}\,[\text{fm_insert/insert, fm_remove/remove,} \\ &\qquad\qquad \text{fm_empty/empty}] \end{aligned}$$

where the nine associated actions are all hidden in the top-level agent.

Now, the four agents corresponding to processes can be defined. At any time, any number of user processes could be sending mail messages to the mail process by placing data in the CollectMail buffer. This dynamic behaviour is modelled in a more static manner by having a single collecting agent that can repeatedly collect posted mail messages (and can do nothing else). Its CCS definition is:

$$\text{Collect} \stackrel{\text{def}}{=} \text{letter_posted}\,.\,\overline{\text{cm_insert}}\,.\,\text{Collect}$$

where letter_posted is the input action for receiving a mail message posted by a user.

A new delivery process was started by the mail process every time a message had to be delivered, and then the mail process waited until this process terminated before proceeding. This dynamic behaviour is modelled in a more static manner by having a single delivery agent that can repeatedly deliver mail messages (and can do nothing else) and which interacts with the mail agent via a 'start' and a 'stop' action for every message. A mail message could actually be sent to a user or, if the user had set mail forwarding to one or more other computer systems, could cause the generation of one or more mail messages to be sent back to the network. This is modelled by the delivery agent incorporating appropriate internal non-determinism; as a simplification for now, forwarding is assumed to generate only one outgoing mail message. The agent's CCS definition is:

$$\begin{aligned} \text{Deliver} \stackrel{\text{def}}{=} \text{md_start}\,.\,(&\tau\,.\,\overline{\text{letter_deliver}}\,.\,\overline{\text{md_stop}}\,.\,\text{Deliver} \\ &+ \tau\,.\,\overline{\text{cm_insert}}\,.\,\overline{\text{md_stop}}\,.\,\text{Deliver}) \end{aligned}$$

where $\overline{\text{letter_deliver}}$ is the output action for sending a mail message to a user.

The mail process was responsible for handling all matters related to the 'Grey Book' electronic mail protocol. It handled both incoming mail files from the FileMail buffer and outgoing mail messages from the CollectMail buffer. The detailed behaviour of the process was that, after handling either an incoming file or an outgoing message, all remaining messages in the CollectMail buffer were handled; this was meant to ensure that user processes posting mail are not

unduly delayed. (In theory, such behaviour could lead to 'starvation' affecting the handling of incoming files but, in practice, this is not likely to happen.) When the mail process handled a mail file from the FileMail buffer, it either could generate a mail message to be delivered to a user or, if the file was invalid in some way, could generate an error-reporting mail file to be sent back to the network. This is modelled by the mail agent incorporating appropriate internal non-determinism. Its CCS definition (using an auxiliary agent Mail1) is:

$$\begin{aligned}\mathrm{Mail} &\stackrel{\mathrm{def}}{=} \mathrm{fm_remove}.(\tau.\overline{\mathrm{md_start}}.\mathrm{md_stop}.\mathrm{Mail1} \\ &\qquad\qquad + \tau.\overline{\mathrm{mf_insert}}.\mathrm{Mail1}) \\ &\quad + \mathrm{cm_remove}.\overline{\mathrm{mf_insert}}.\mathrm{Mail1} \\ \mathrm{Mail1} &\stackrel{\mathrm{def}}{=} \mathrm{cm_empty}.\mathrm{Mail} + \mathrm{cm_remove}.\overline{\mathrm{mf_insert}}.\mathrm{Mail1}\end{aligned}$$

where all of the actions are internal to the top-level agent. The Mail1 agent makes use of the cm_empty action to check whether or not all mail messages in the CollectMail buffer have been handled.

Finally, the file transfer process was responsible for handling all matters related to the 'Blue Book' file transfer protocol. It handled both incoming files received from the network and outgoing files from the MailFile buffer. The detailed behaviour of the process meant that it could handle an arbitrary number of files received from the network between each handling of a file from the MailFile buffer; this was a unintentional feature of the internal organisation of the implementation that was potentially dangerous, since accepting new work was given priority over ridding the system of completed work. (Again, in theory, such behaviour could lead to 'starvation' affecting the handling of files from the MailFile buffer but, in practice, this is not likely to happen.) The behaviour can be precisely modelled by an agent that just embodies simple non-determinism. Its CCS definition is:

$$\mathrm{File} \stackrel{\mathrm{def}}{=} \mathrm{file_received}.\overline{\mathrm{fm_insert}}.\mathrm{File} + \mathrm{mf_remove}.\overline{\mathrm{file_send}}.\mathrm{File}$$

where file_received is the input action for receiving a file from the network, and $\overline{\mathrm{file_send}}$ is the output action for sending a file to the network.

The CCS description of the composition of the above sub-agents was supplied to the CWB, and the behaviour of the overall mail system model was investigated. The model-checking facility confirmed that the overall agent would not always evolve to a deadlock-free agent. Further, the deadlock-finding facility revealed the extra information that evolution was possible to any of the following three deadlocked agents:

$$\begin{aligned}\mathrm{Dead1} \stackrel{\mathrm{def}}{=} &(\overline{\mathrm{cm_insert}}.\mathrm{Collect} \\ &\mid \mathrm{Deliver} \\ &\mid \overline{\mathrm{cm_remove}}.\mathrm{CollectMail} \\ &\mid \overline{\mathrm{mf_insert}}.\mathrm{Mail1} \\ &\mid \overline{\mathrm{mf_remove}}.\mathrm{MailFile} \\ &\mid \overline{\mathrm{fm_remove}}.\mathrm{FileMail} \\ &\mid \overline{\mathrm{fm_insert}}.\mathrm{File}) \\ &\backslash\{\ldots\}\end{aligned}$$

$$
\begin{aligned}
\text{Dead2} \stackrel{\text{def}}{=} \; & (\overline{\text{cm_insert}}.\text{Collect} \\
& \mid \overline{\text{cm_insert}}.\overline{\text{md_stop}}.\text{Deliver} \\
& \mid \overline{\text{cm_remove}}.\text{CollectMail} \\
& \mid \text{md_stop}.\text{Mail1} \\
& \mid \text{MailFile} \\
& \mid \overline{\text{fm_remove}}.\text{FileMail} \\
& \mid \overline{\text{fm_insert}}.\text{File}) \\
& \backslash \{\ldots\}
\end{aligned}
$$

$$
\begin{aligned}
\text{Dead3} \stackrel{\text{def}}{=} \; & (\overline{\text{cm_insert}}.\text{Collect} \\
& \mid \overline{\text{cm_insert}}.\overline{\text{md_stop}}.\text{Deliver} \\
& \mid \overline{\text{cm_remove}}.\text{CollectMail} \\
& \mid \text{md_stop}.\text{Mail1} \\
& \mid \overline{\text{mf_remove}}.\text{MailFile} \\
& \mid \overline{\text{fm_remove}}.\text{FileMail} \\
& \mid \overline{\text{fm_insert}}.\text{File}) \\
& \backslash \{\ldots\}
\end{aligned}
$$

which shows that two different kinds of deadlock are possible. These can be interpreted in terms of the software being modelled. In the case of Dead1, there is a deadlock between the mail and file transfer processes, since both the MailFile and FileMail buffers are full, and the two processes are attempting to send files to each other (a collection process is also unable to proceed, but this does not contribute to the deadlock). In the case of Dead2 and Dead3, there is a deadlock between a delivery process and the mail process, since the delivery process is attempting to send a forwarded mail message to the mail process, but the CollectMail buffer is full and the mail process is waiting for the delivery process to stop. The only difference between Dead2 and Dead3 is whether or not the MailFile buffer is full; this feature does not contribute to the deadlock.

The real observed deadlock corresponded to Dead1; the other two deadlocks indicated a potential problem that had not been encountered in practice. For each deadlocked agent, the deadlock-finding facility also reported a specimen sequence of actions which would cause evolution to that agent. Although interesting, this information did not shed additional light on the real deadlock: the suggested sequence for Dead1 (containing two file_received actions followed by four letter_posted actions) seemed unlikely to have led to the observed problem. In order to investigate the causes of both types of deadlock further, more experiments were performed, and these are described in the next section.

4. Experiments on the Nature of the Deadlock

The mail system model incorporated the (correct) assumption that electronic mail could both be arriving from the network and also posted by users. The cause of the deadlocks was investigated in more detail by checking the behaviour of the model when each of these activities happened in isolation.

Arrival of mail from the network can be eliminated from the model by

changing the File agent to be just:

$$\mathrm{File} \stackrel{\mathrm{def}}{=} \mathrm{mf_remove}.\overline{\mathrm{file_send}}.\mathrm{File}$$

and, given this, the revised system was found to be deadlock-free. The explanation is that the only remaining inter-process data flow is from user processes via the mail process to the file transfer process, with no looping possible — the mail and file transfer processes just act as extra buffers within the overall system.

Posting of mail by users can be eliminated from the model just by removing the Collect agent from the overall system. The revised model was found still to contain the same potential for deadlock as before. This was reassuring because, in the case of the real observed deadlock, there had been no evidence that any users had been sending mail around the time that the problem occurred.

After these two initial experiments, an examination of the inter-process data flows possible revealed two obvious ways in which looping (and thence deadlock) might occur: the automatic generation of error-reporting mail by the mail process, and the automatic forwarding of mail by a delivery process. To investigate the effect of these features, the next experiments involved checking the mail system when one or both were suppressed.

Automatic generation of error-reporting mail can be eliminated from the model by changing the Mail agent to be:

$$\begin{aligned} \mathrm{Mail} &\stackrel{\mathrm{def}}{=} \mathrm{fm_remove}.\overline{\mathrm{md_start}}.\mathrm{md_stop}.\mathrm{Mail1} \\ &\quad + \mathrm{cm_remove}.\overline{\mathrm{mf_insert}}.\mathrm{Mail1} \\ \mathrm{Mail1} &\stackrel{\mathrm{def}}{=} \mathrm{cm_empty}.\mathrm{Mail} + \mathrm{cm_remove}.\overline{\mathrm{mf_insert}}.\mathrm{Mail1} \end{aligned}$$

but this revised system was found to still have the same potential for deadlock (regardless of whether or not the posting of mail by users was also suppressed). Again, this was reassuring because, in the case of the real observed deadlock, there had been no evidence that any error-reporting mail had been generated.

Automatic forwarding of mail can be eliminated from the model by changing the Deliver agent to be:

$$\mathrm{Deliver} \stackrel{\mathrm{def}}{=} \mathrm{md_start}.\overline{\mathrm{letter_deliver}}.\overline{\mathrm{md_stop}}.\mathrm{Deliver}$$

and, with this modification, a change in system behaviour was discovered (regardless of whether or not the posting of mail by users was also suppressed). The revised system had potential for deadlock, but only in terms of evolution to the Dead1 agent, i.e., deadlock between the mail and file transfer processes. This experiment indicated that automatic forwarding of mail definitely was a mail system feature that impacted on deadlock-proneness, albeit in a manner which differed from its suspected role in leading to the real observed deadlock.

After the experimentation concerning possible deadlock causes, two more experiments were conducted to discover whether simple adjustments to scheduling within the mail or file transfer processes could eliminate deadlock-proneness. The first experiment involved modifying the behaviour of the mail process so that it no longer handled all of the available mail messages in the CollectMail buffer before dealing with any mail files from the FileMail buffer (and so tending to increase the total amount of work within the system). The revised Mail agent is:

$$\begin{aligned} \mathrm{Mail} &\stackrel{\mathrm{def}}{=} \mathrm{cm_remove}.\mathrm{Mail1} + \mathrm{fm_remove}.\mathrm{Mail2} \\ \mathrm{Mail1} &\stackrel{\mathrm{def}}{=} \overline{\mathrm{mf_insert}}.(\mathrm{fm_empty}.\mathrm{Mail} + \mathrm{fm_remove}.\mathrm{Mail2}) \end{aligned}$$

$$\text{Mail2} \stackrel{\text{def}}{=} \tau . \overline{\text{md_start}} . \text{md_stop} . \text{Mail3} + \tau . \overline{\text{mf_insert}} . \text{Mail3}$$

$$\text{Mail3} \stackrel{\text{def}}{=} \text{cm_empty} . \text{Mail} + \text{cm_remove} . \text{Mail1}$$

which handles the two buffers in strict alternation when both contain data. The revised system was still prone to the same kinds of deadlock as the original system. However, deadlock was only possible when sending of mail by users was enabled and so, apparently, the real observed deadlock would have been avoided had this scheduling modification been made. Such an illusion could be rapidly shattered, however, by making the model a little more realistic: as noted earlier, automatic forwarding only resulted in one mail file being sent back to the network but, in practice (as with the Dr Who Fan Club), it could result in more than one file because mail might be forwarded to several different destinations. If the above Mail agent was adjusted again so that it non-deterministically placed either one or two files in the MailFile buffer when forwarding, the change was sufficient to make deadlock between the mail and file transfer processes possible again.

The second experiment involved modifying the file transfer process so that it no longer handled an arbitrary number of files received from the network before dealing with each file in the MailFile buffer (and so tending to increase the total amount of work within the system). The revised File agent is:

$$\text{File} \stackrel{\text{def}}{=} \text{mf_remove} . \overline{\text{file_send}} . \text{File} + \text{file_received} . \overline{\text{fm_insert}} . \text{File1}$$

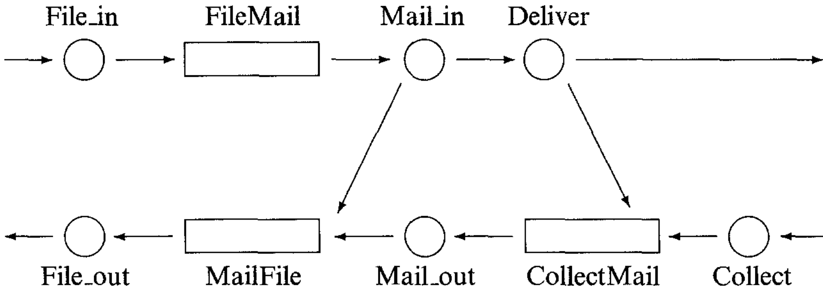
$$\text{File1} \stackrel{\text{def}}{=} \text{mf_empty} . \text{File} + \text{mf_remove} . \overline{\text{file_send}} . \text{File1}$$

which handles all files in the MailFile buffer after each file has been received from the network. Although incorporating more sensible internal scheduling, this revised system did not have any better behaviour from a deadlock point of view, since it still had the potential for the Mail process to attempt to place more than one file in the MailFile buffer when the File process was already waiting with a new file received from the network.

After conducting these experiments, it was clear that it was not possible to remove deadlock-proneness just by altering the behaviour of the existing mail and file transfer processes, and that rather more significant modification of the mail system implementation was required. The final solution adopted involved a reorganisation of the system processes and their interactions, and it is described in the next section.

5. The Revised Mail System

To remove the deadlock between the delivery and mail processes, it was decided to partition the mail process into two separate new processes, so that mail files from the FileMail buffer were handled by one process, and mail messages from the CollectMail buffer were handled by the other process. Similarly, to remove the deadlock between the mail and file transfer processes, it was decided to partition the file transfer process into two new processes, so that files received from the network were handled by one process, and files from the MailFile buffer were handled by the other process. As well as removing deadlock potential, such a restructuring appeared better from the point of view of enhancing modularity in the overall software system; those functions that were common to both of a new pair of processes, principally protocol handling functions, were placed in shared libraries. The new mail system is shown below:



It can be modelled by a new CCS agent incorporating four new agents in place of two old ones. The overall definition is:

(Collect
| CollectMail
| Deliver
| Mail_in | Mail_out
| MailFile
| FileMail
| File_in | File_out)
\ {cm_insert, cm_remove, cm_empty,
md_start, md_stop,
mf_insert, mf_remove, mf_empty,
fm_insert, fm_remove, fm_empty}

The new sub-agents (three of them very simple) are:

$$\mathrm{Mail_in} \stackrel{\mathrm{def}}{=} \mathrm{fm_remove}\,.\,(\tau\,.\,\overline{\mathrm{md_start}}\,.\,\mathrm{md_stop}\,.\,\mathrm{Mail_in} + \tau\,.\,\overline{\mathrm{mf_insert}}\,.\,\mathrm{Mail_in})$$

$$\mathrm{Mail_out} \stackrel{\mathrm{def}}{=} \mathrm{cm_remove}\,.\,\overline{\mathrm{mf_insert}}\,.\,\mathrm{Mail_out}$$

$$\mathrm{File_in} \stackrel{\mathrm{def}}{=} \mathrm{file_received}\,.\,\overline{\mathrm{fm_insert}}\,.\,\mathrm{File_in}$$

$$\mathrm{File_out} \stackrel{\mathrm{def}}{=} \mathrm{mf_remove}\,.\,\overline{\mathrm{file_send}}\,.\,\mathrm{File_out}$$

and these have an obvious correspondence to the Mail and File sub-agents of before (the 'empty buffer' action is no longer needed).

The new mail system model was supplied to the CWB. Both the deadlock-finding facility and the model-checking facility reported that deadlock was not possible. Although the CCS model embodies certain simplifications, such as smaller buffer capacities and the fact that only one file is sent to the network for each mail message generated, it is reasonably clear by considering the straightforward inter-process data flows in the new system that more sophisticated modelling would not reveal any potential for deadlock.

The processes implementing the real electronic mail system were restructured in the above way, and no further deadlock problems were encountered, in particular during subsequent night-time bombardments by the information-laden epistles of the Dr Who Fan Club.

6. Conclusions

With the emergent availability of tools to assist in the analysis of concurrent systems, an increasing number of real-life systems are being modelled and verified effectively. The most common type of verification criterion is that a modelled system exhibits behaviour which is equivalent to the behaviour of a simpler system that is 'obviously correct'; for example, that a point-to-point communication protocol has behaviour equivalent to a simple buffer. In other cases, particularly where only some aspects of overall behavour are of interest, the verification criterion is that a modelled system satisfies a specification formulated in a modal logic; this style of verification was used here, in addition to checking satisfaction of a 'hard wired' specification of deadlock-freedom.

It is necessary to abstract the essential features of a concurrent system in order to produce a model for which analysis is tractible; experience here has been that it is non-trivial to ensure that such abstraction is accurate, and that it does not omit essential details. Given that a suitable model can be obtained, this work has confirmed that it is very desirable to perform full and formal analysis, as opposed to thinking about behaviour informally, or even to just observing how a real system behaves. Unforeseen deadlock possibilities were discovered, and the experiments in section 4 sometimes revealed behaviour that was not immediately obvious when thinking informally.

Although the current version of the Edinburgh Concurrency Workbench, like other similar tools under development, is irritating because of its inefficient implementation and its rather primitive user interface, there is no doubt that its functional capabilities are extremely useful.

Acknowledgement

I thank Jo Blishen, Stephen Gilmore and Faron Moller for helpful comments.

References

[BdR90] Boudol, G., de Simone, R., Roy, V. and Vergamini, D.: Process Calculi, from Theory to Practice: Verification Tools. *Springer-Verlag LNCS 407*, pp. 1–10, 1990.

[Bou85] Boudol, G.: Notes on Algebraic Calculi of Processes. In Apt (ed) Logics and Models of Concurrent Systems. Springer-Verlag, 1985.

[Cle90] Cleaveland, R., Parrow, J. and Steffen, B.: The Concurrency Workbench. *Springer-Verlag LNCS 407*, pp. 24–37, 1990.

[InP91] Inverardi, P. and Priami, C.: Evaluation of Tools for the Analysis of Communicating Systems. *Bulletin of the EATCS no. 45*, pp. 158–185, 1991.

[Koz83] Kozen, D.: Results on the Propositional Mu-calculus. *Theoretical Computer Science*, 27, 333–354 (1983).

[LGZ89] Larsen, K., Godskesen, J. and Zeeberg, M.: TAV, Tools for Automatic Verification, User Manual, Technical Report R 89-19, Dept of Mathematics and Computer Science, Ålborg University, 1989.

[Mar91] Marsden, B.: Communication Network Protocols: OSI Explained. Chartwell-Bratt, 1991.

[Mil89] Milner, R.: Communication and Concurrency. Prentice-Hall, 1989.

[Mor90] Morley, M.: Tactics for State Space Reduction on the Concurrency Workbench, Technical Report ECS-LFCS-90-109, Dept of Computer Science, University of Edinburgh, 1990.

[Sti89] Stirling, C.: Temporal Logics for CCS, *Springer-Verlag LNCS 354*, pp. 660–675, 1989.

[Tan88] Tanenbaum, A.: Computer Networks. Prentice-Hall, 1988.

Table 1. Execution times for original CCS descriptions

Mail system	"fd" time	"cp" time
Original	414	28
New	4569	1740

Table 2. Execution times for revised CCS descriptions

Mail system	"fd" time	"cp" time
Original	177	13
New	2331	1533

A. Details of Concurrency Workbench Use

Version 6.11 of the Edinburgh Concurrency Workbench (CWB) was run on a Sun 4/690 computer with 128 megabytes of memory. For each agent analysed, the deadlock-finding facility (the CWB "fd" command) was used, and the model-checking facility (the CWB "cp" command) with the proposition "max(X.min(Y.<-t>T | <->Y) & [-]X)" was used. The approximate CPU time in seconds required for each test is shown in Table 1. (Note that there was no significant difference in execution time if the simpler modal logic deadlock-freedom specification "max(X.<->T & [-]X)" was checked instead.)

Using a further CWB facility, the "size" command, it was discovered that the original mail system model could evolve to any of 718 distinct agents, and that the new mail system model could evolve to any of 1613 distinct agents; this accounts for the increased time to investigate the new mail system. However, further investigation revealed that many pairs of 'distinct' agents were only syntactically distinct: for example, agents incorporating the identifier "CollectMail" were distinguished from other agents which were identical, except for incorporating "Buffer [cm_insert/insert,cm_remove/remove, cm_empty/empty]" instead.

To avoid this problem, the auxiliary definitions of CollectMail, MailFile and FileMail were removed from the mail system models. The revised agents were reported as being able to evolve to only 405 and 1024 distinct descendent agents respectively. With this revision, the CPU time in seconds required for each test is shown in Table 2. There is a significant improvement, without any loss of function.

To fully explain and discuss the above timings, knowledge of the internal operation of the CWB is necessary: such detail is outwith the scope of this paper. The absolute times are not especially important, since it is acknowledged that the present version of the CWB is experimental, and has not been implemented with an emphasis on efficient execution. The deadlock-finding facility is always slower than the model-checking facility for at least two reasons: first, it checks *all* descendent agents for deadlock-proneness; and second, for each deadlocked agent, it finds and displays a specimen sequence of actions which leads to that agent. The model-checking facility needs only to examine sufficiently many agents in order to determine a true/false result; here, this involves only finding one deadlocked descendent agent in the case of the original mail system, but involves checking all descendent agents in the case of the new mail system.

Received July 1992
Accepted in revised form January 1993 by J. Parrow