DIVISION OF COMPUTER SCIENCE

A Program for Animating CCS Specifications

Technical Report No.145

J. R. Stobo

October, 1992

A Program for Animating CCS Specifications

J. R. Stobo
University of Hertfordshire
College Lane
Hatfield, AL10 9AB
email: comqjrs@herts.ac.uk

October 12, 1992

Abstract

This report describes the operation of a program which enables the behaviours admitted by a specification in the Calculus of Communicating Systems (CCS) to be investigated. Two specifications in the calculus are presented to illustrate the issues under discussion, and program output from the animation of these specifications is shown in some detail, both to make clear the program's value and limitations and to highlight distinctive features of specifications in CCS.

A full description of how to use the program is given in Sections 2.2 and 3.1, so these parts of the report may serve as a user manual for the program.

Contents

1 Introduction

A specification of a system in CCS [Mil 80,Mil 89] consists of a description of the component processes of the system, called *agents*, which is given in terms of the communication events in which the agents may engage, together with an agent expression for the system as a whole, usually defined as the *parallel composition* of two or more of the component agents. An agent may engage in a communication action with another agent in the system or communicate with the system's environment. Events of the former type are internal to the system, and are not directly observable from outside it.

Save when engaging in an internal communication event, agents in a system proceed in their actions entirely independently of each other. A specification does not define an order for the actions of distinct agents in communicating with the system's environment. For this reason, a CCS specification admits many possible behaviours for an implementation of the system, those behaviours corresponding to the different actual sequencings of the external communication actions of the agents in the system.

The *expansion law* of CCS defines what linearisations of actions a given specification admits, making it possible to investigate the behaviours admitted by the specification. Naturally, the purpose of such an investigation is to discover, by repeated applications of the expansion law, whether there are any undesirable sequences of actions that meet the specification. However, each sequence may have many alternative next actions at every step, and the number of distinct sequences of actions that the specification admits grows very rapidly as each is extended by successive applications of the law. For any but the most trivial specification, this combinatorial explosion of sequences makes application of the expansion law by hand very error-prone and prohibitively slow.

This report describes a program which automates the law, thus overcoming the human propensity for error in its application to complex CCS expressions. Section 2 gives an informal description of the expansion law, and then illustrates how CCS agent definitions are input to the program and the style of presentation of the result of applying the law to them, which is the program's output.

It is characteristic of a CCS specification that most sequences of actions it admits manifestly are correct, in terms of the requirements upon the system, and just a few require close investigation. For this reason, the program enables the expansion law to be applied just to a selected part of a CCS expression, representing the suspect sequence. Other parts of the expression, presumed to describe recognisably safe sequences of actions, are dropped from the resultant expanded expression. The use of this program feature is described in Section 3.1.

By allowing just a portion of a CCS expression to be selected for expansion, the program offers a partial solution to the problem of combinatorial explosion. Another method, which is intended to be implemented as a future extension to the program, is the provision of a graphical interface to the program. This is described in Section 3.2.

Besides allowing us to investigate the properties of a CCS specification, the program also eluci-

dates characteristics of the specification notation itself. So, in Section 4 we evaluate features of CCS, on the basis of our experience of specifying an unbounded first-in, first-out queue in CCS, and of investigating it using the program.

2 Automating the Expansion Law

2.1 Meaning of the Expansion Law

An expression for a CCS agent in which there are no alternatives is defined recursively to consist of a *prefix*, representing the first communication action of the agent, followed by an agent expression. In its simplest form, the expansion law formalises the notion that the first action of a system which consists of two or more such agents composed in parallel may be the first action of any one of those agents. Thus, for agents M and N, where:

$$M \stackrel{\text{def}}{=} \alpha \cdot \beta \cdot M$$
$$N \stackrel{\text{def}}{=} \gamma \cdot \delta \cdot N$$

the law states:

$$M \mid N = \alpha \cdot (\beta \cdot M \mid N) \; + \; \gamma \cdot (M \mid \delta \cdot N)$$

Here "|" denotes parallel composition of agents and "+" denotes alternatives and is called the *summation* operator. The brackets are necessary because the summation operator binds more tightly than the parallel composition operator. The prefixing operator "$\cdot$" binds most tightly. The expression after the application of the law is to be read as stating that the first action of a system comprising agents M and N composed in parallel may be action α or action γ. In the first case, the behaviour of the system after α is defined to be $\beta \cdot M \mid N$, and in the second case the behaviour after γ is defined as $M \mid \delta \cdot N$. As one would expect, expressions in each alternative for those agents whose first action has not occurred are unchanged by the application of the law.

When the composed actions themselves contain alternatives, the law defines:

$$\alpha \cdot A \; + \; \beta \cdot B \mid \gamma \cdot M = \alpha \cdot (A \mid \gamma \cdot M) \; + \; \beta \cdot (B \mid \gamma \cdot M) \; + \; \gamma \cdot (\alpha \cdot A \; + \; \beta \cdot B \mid M)$$

This is also intuitively appealing: after a system has carried out the communication action that prefixes one of several alternatives, the actions available are just those in the expression that followed the particular prefix, and the actions that were prefixes of other alternatives are no longer available. That the system is committed in each alternative to one or other action can be illustrated by drawing a *derivation tree*. In Figure 1, the labels on branches represent actions and expressions at nodes define states of the system:

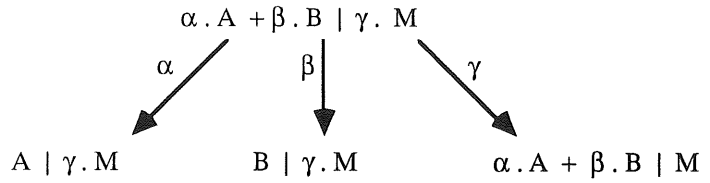

Figure 1 Derivation Tree Showing Alternative First Actions

The law can be applied to an expression that consists of alternatives rather than a parallel composition, provided each alternative is a parallel composition. This is the means by which the sequences of alternative actions may be teased out from the specification. A second expansion of the expression $M \mid N$, with agents M and N defined as above, yields the expression:

$$\alpha \cdot (\beta \cdot (M \mid N) \; + \; \gamma \cdot (\beta \cdot M \mid \delta \cdot N)) \; + \; \gamma \cdot (\alpha \cdot (\beta \cdot M \mid \delta \cdot N) \; + \; \delta \cdot (M \mid N))$$

which corresponds to the derivation tree:

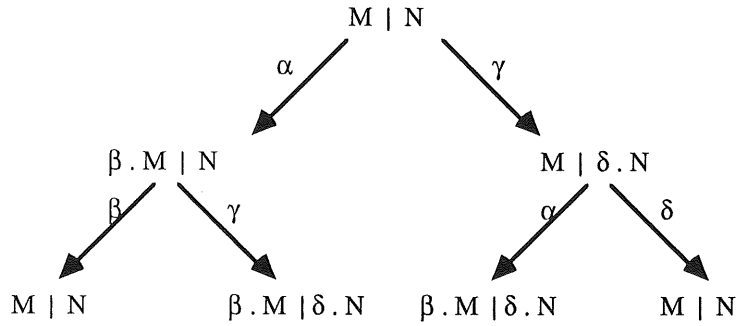

Figure 2 Derivation tree after two actions

In fact, this tree is a graph, with the state $\beta \cdot M \mid \delta \cdot N$ reachable by two different sequences of actions and the two occurrences of state $M \mid N$ being duplicates of the system's initial state.

Finally, the expansion law defines the effect of internal communication between composed agents of the system. Internal communication may occur whenever prefixes of agents are *complementary*: they are of the form x and $\overline{x}$. These two forms represent inputs to and outputs from the two agents, though in the absence of any passing of data values between agents, it is more accurate to think of the internal communication as a handshaking, a straightforward synchronisation of the actions of the two agents. A specification which makes use of the value-passing calculus is the FIFO of Section 4. For complementary actions, the form of the law is:

$$\alpha \cdot \mathcal{A} \mid \overline{\alpha} \cdot \mathcal{B} = \alpha \cdot (\mathcal{A} \mid \overline{\alpha} \cdot \mathcal{B}) \; + \; \overline{\alpha} \cdot (\alpha \cdot \mathcal{A} \mid \mathcal{B}) \; + \; \tau \cdot (\mathcal{A} \mid \mathcal{B})$$

The symbol "τ" denotes an internal communication action between two components of the system. The action has no direct effect on the system's environment, and from the point of view of the environment, all internal communication actions are identical.

The possibility that an internal communication event may occur does not preclude external communication. The *restriction* operator "\" may be used to prohibit an action from being an external event. If we have:

$$(\alpha \cdot A \mid \overline{\alpha} \cdot B) \setminus \{\alpha\}$$

the law defines the expansion to be simply:

$$\tau \cdot (A \mid B)$$

The *renaming* operator allows otherwise distinct prefixes in the expressions for different agents to be made complementary when those agents are composed in parallel. So, if agent P produces $\overline{output1}$ and agent Q receives $input1$, as follows:

$$P \stackrel{\text{def}}{=} \overline{output1} \cdot R$$
$$Q \stackrel{\text{def}}{=} input1 \cdot S$$

where agents R and S are defined elsewhere in the specification, we can indicate that in the total system, Q receives its first input from P by renaming either $\overline{output1}$ or $input1$ to be complementary to the other, or, more perspicuously, by renaming both to indicate the role of the communication event in the total system:

$$P[throughput1/output1] \mid Q[throughput1/input1]$$

The expansion of this expression is then:

$$\overline{throughput1} \cdot (R \mid throughput1 \cdot S) \; + \; throughput1 \cdot (\overline{throughput1} \cdot R \mid S) \; + \; \tau \cdot (R \mid S)$$

If we intended P to send its output only to Q, and Q to receive its input only from P, we would restrict the renamed action:

$$(P[throughput1/output1] \mid Q[throughput1/input1]) \setminus \{throughput1\}$$

The inclusion of a prefix in the restriction set causes both it and its complement to be restricted.

2.2 Using the Program

The program is written in Prolog within the LPA Prolog environment on the Macintosh.[1] It supports all the aspects of the CCS notation and the semantics of the expansion law that were described in Section 2.1.

When the program is started, a `CCS` menu appears on the menu bar, with selections **`Define`**, **`Expand`**, **`Agents`**, **`Display`**, **`Load`** and **`Save`**. If agent definitions input previously had been **`Saved`**

[1] It is intended at a later date to convert the program to a standalone clickable application, to obviate the need for users to have access to the LPA environment.

to a file, they could now simply be `Loaded` from it. Alternatively, selection of the `Define` option brings up a dialogue box:

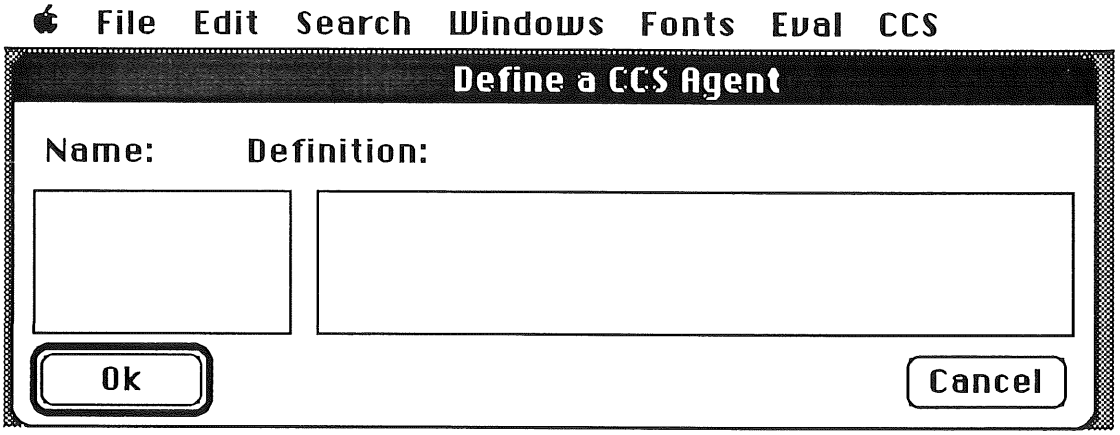

Figure 3 Dialogue Box for Agent Definition

The syntax of agent names and of agent expressions is exactly that of CCS, as described in Section 2.1, except that agent names and prefixes must begin with a lower-case letter and that instead of the bar to indicate an output, we prefix output actions with "!".[2] So, we might enter:

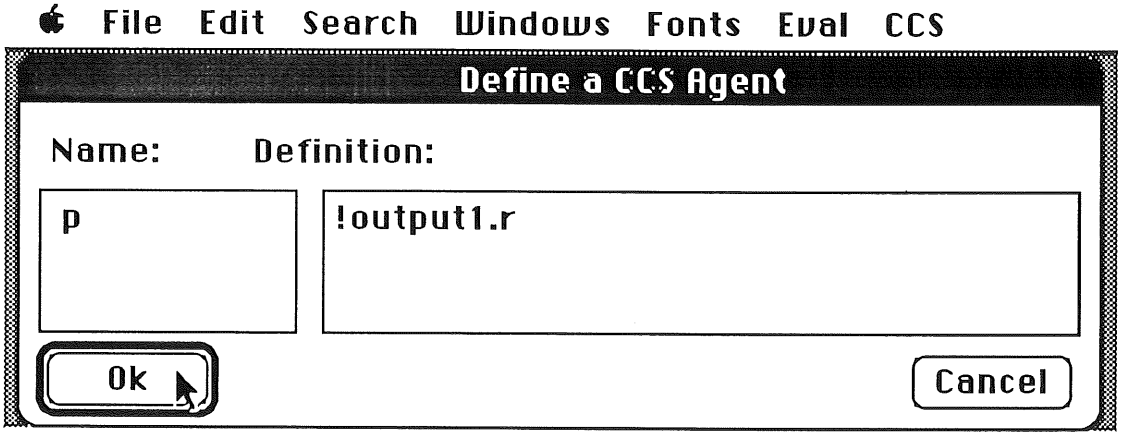

Figure 4 Defining the Agent p

After also inputting the definition of agent q that was given in Section 2.1, we might wish to check the definitions. Selecting the `Display` option causes the agents and their defining expressions to be listed in a window called `Definitions`. Alternatively, the `Agents` option brings up a scrolling dialogue box showing the names of all agents known to the program. When one of these names is selected, the corresponding definition is displayed in the dialogue box.

Once satisfied that agents p and q are defined as intended, we would then select `Expansion` to

[2] By analogy with the notation for an output variable in a Z schema.

know the first action of the composed, renamed and restricted system:

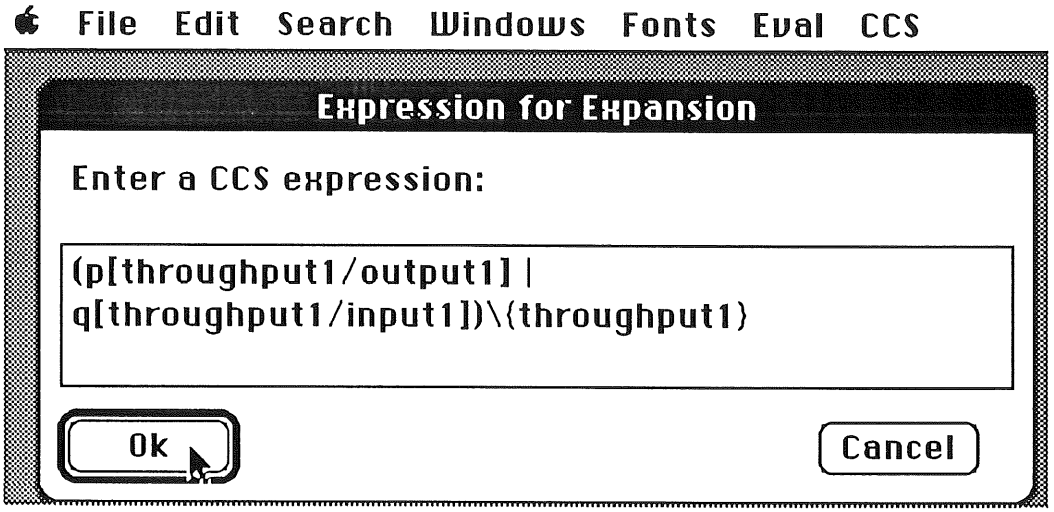

Figure 5 Dialogue Box for Expression Expansion

Alternatively we might use the `Define` option to name this expression, say as t, and if so we would just type that name in the dialogue box of Figure 5.

When the user clicks on `Ok` or presses "Return", the expression for the system after one application of the expansion law is displayed on screen in a window called `Expansion`. In the absence of definitions for agents r and s, expansion of the expression shown in Figure 5 could only be carried out once. To show the style of display of larger expressions, we revert to the example of the system formed from the parallel composition of the agents M and N that were defined in Section 2.1. The display after two applications of the law is shown in Figure 6. After each application, the `Continuation` dialogue box appears, as shown in the Figure, and if the user clicks on `Ok` or presses "Return", the next expansion is shown beneath the previous one in the `Expansion` window.

Alternatives are shown below each other, so we see:

```
c.(a.b.m | d.n) +
a.(b.m | c.d.n)
```

as the result of the first expansion. At subsequent expansions, each alternative is likely to generate further alternatives, these representing branches at lower levels of the derivation tree. These nested alternatives are also shown beneath each other, indented to indicate their status as sub-alternatives.

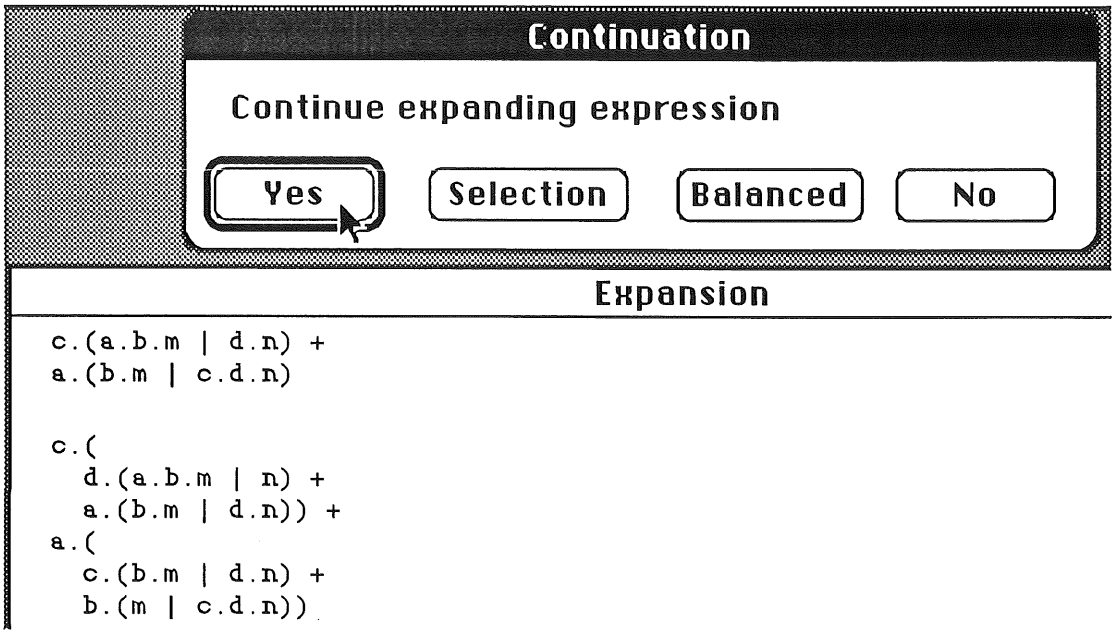

Figure 6 Program Output after Two Applications of the Expansion Law to the Expression $M \mid N$

When a specification includes renaming of prefixes or their restriction, the action of these operators is reflected in the expressions displayed, though the operators themselves are not displayed. To show them would make the display rather cluttered, and the user should picture renamings as immediately following every occurrence in the expression of the agent name they qualify and restrictions as enclosing the whole expression.

If a specification consisted of definitions of agents built up in a number of layers, the situation might arise in which restrictions were nested inside other restrictions, as in the following set of definitions:

$$
\begin{aligned}
p &\stackrel{\text{def}}{=} a \cdot b \cdot p \\
q &\stackrel{\text{def}}{=} \overline{a} \cdot d \cdot q \\
r &\stackrel{\text{def}}{=} e \cdot \overline{d} \cdot r \\
s &\stackrel{\text{def}}{=} \overline{e} \cdot h \cdot s \\
i &\stackrel{\text{def}}{=} (p \mid q) \setminus \{a\} \\
j &\stackrel{\text{def}}{=} (r \mid s) \setminus \{e\} \\
k &\stackrel{\text{def}}{=} (i \mid j) \setminus \{d\}
\end{aligned}
$$

The program accepts nested applications of the restriction operator. However, it treats all restrictions as applying to a whole expression, even if initially attached to a sub-expression which is just one summand or composand. In this respect, the program does not fully implement the semantics of the restriction operator given in the *Restriction Laws* [Mil 89, page 80], where a side-condition defines the circumstances in which applications of the operator may be moved

outwards. To the program, the name k in the above fragment denotes the agent expression:

$$(a \cdot b \cdot p \mid \overline{a} \cdot d \cdot q \mid e \cdot \overline{d} \cdot r \mid \overline{e} \cdot h \cdot s) \setminus \{a, e, d\}$$

Indeed, it is into this form, called *standard concurrent form* in [Mil 89], that the program transforms every expression typed into the **Expansion** dialogue box, before actually applying the expansion law.

A similar situation might arise in the case of the renaming operator, but it is not at all clear what meaning could be ascribed to an expression which, prior to transformation into standard concurrent form, included nested renamings. The renamings could not simply be taken all to apply to the whole expression, for a prefix might be renamed to different names in different parts of it. Yet, if the nesting was preserved, different standard concurrent forms might be derivable according to the order in which renamings were applied. In the face of this uncertainty, and in the absence of any statement on the subject in the *Relabelling Laws* of [Mil 89], a decision was taken to prohibit such structures, and if the program detects this pathological situation in the course of computing the standard concurrent form of a given expression, it issues a warning message and halts.

3 Applying the Law to Sub-expressions

3.1 The First Illustrative Specification

As was explained in Section 1, the need to be able to expand an expression selectively arises when one is faced with a combinatorial explosion of sequences of system actions represented in a complex CCS expression. In the interests of clarity, however, we shall illustrate this feature of the program by reference to a specification which is, in fact, sufficiently simple for the feature to be scarcely necessary, that of a primitive vending machine. Nonetheless it is complex enough to illustrate the point at issue.

The machine is required to accept a 5p coin and deliver a cup of tea and 1p as change. The specification is built by composing a change-giver:

$$cg \stackrel{\text{def}}{=} in5 \cdot \overline{out2} \cdot \overline{out2} \cdot \overline{out1} \cdot cg$$

with a simple vending machine:

$$vms \stackrel{\text{def}}{=} in2 \cdot in2 \cdot \overline{outtea} \cdot vms$$

as follows:

$$s \stackrel{\text{def}}{=} (cg[\alpha/out2] \mid vms[\alpha/in2]) \setminus \{\alpha\}$$

We may view the specification as defining the system shown in Figure 7.

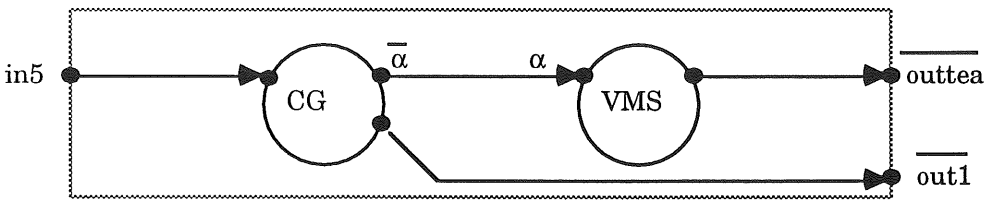

Figure 7 Vending machine constructed from two components

When we input the specification to the program, the first display is:

```
in5.(!a.!a.!out1.cg | vms)
```

indicating that the only first action is to accept 5p from the customer. After five applications of the expansion law, we find the following derivation tree displayed:

```
in5.T.T.(
   !outtea.!out1.(cg | a.a.!outtea.vms) +
   !out1.(
      !outtea.(in5.!a.!a.!out1.cg | vms) +
      in5.(!a.!a.!out1.cg | !outtea.vms)))
```

At this point, we ought to be satisfied as to the correctness of the sequences in which the tea and then the penny or the penny and then the tea are returned. After both, the expression for the system state is equivalent to that for the initial state, indicating that the cycle of actions involved in one purchase has been completed. The third sequence, in which a second 5p is inserted when just the penny has been collected from the previous purchase suggests that the first cup of tea is liable to cost 9p, and it merits further investigation.

The program offers two alternatives. Firstly, we may move the cursor to the **Expansion** window, highlight the sub-expression to be expanded, in this case:

```
!a.!a.!out1.cg | !outtea.vms
```

and then click on `Selected` back in the **Continuation** dialogue box. Alternatively, we may position the cursor anywhere in the desired sub-expression and then click on **Balanced** in the **Continuation** dialogue box. The latter option causes that portion of the expression up to the first matching pair of brackets around the cursor position to be selected for expansion. It can sometimes be useful to see the delimiting brackets around an expression, and by pressing "cloverleaf" and *b*, the sub-expression around the cursor position up to matching brackets is simply highlighted in the **Expansion** window, without application of the expansion law to the portion highlighted.

Whichever method is used for selecting the desired sub-expression, the next display in the **Expansion** window shows just the result of expanding that selection:

```
in5.T.T.!out1.in5.!outtea.(!a.!a.!out1.cg | vms)
```

Notice that the display includes the complete sequence of actions that prefix the expanded sub-expression. It reveals that if the customer does put a second 5p in the machine, the only action then permitted is to take the first cup of tea. We may now conclude that the specification does permit only reasonable behaviours, as the state of the system is now represented by an expression that had occurred at an earlier stage of the investigation, and so all possible sequences of actions have been explored. It will become possible, after the two internal communication events, for the customer once again to take either the tea or the penny dispensed in response to the second 5p.

3.2 An Improved Interface to the Program

The conclusion of Section 3.1 might or might not have been obvious from the expressions shown, depending whether the reader noticed that the last one was the same as that for the state after the very first action. It certainly is obvious from the derivation graph for the system, which is shown in Figure 8.

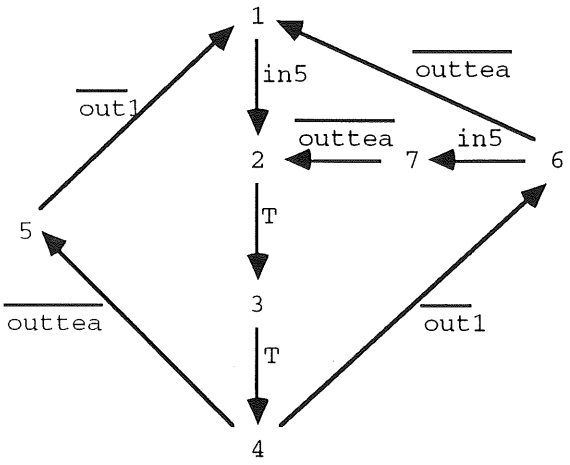

Figure 8 Complete Derivation Graph for the Vending Machine Specification

The expressions for the nodes in the Figure are:

Node 1 = $cg \mid vms$

Node 2 = $\overline{a} \cdot \overline{a} \cdot \overline{out1} \cdot cg \mid vms$

Node 3 = $\overline{a} \cdot \overline{out1} \cdot cg \mid a \cdot \overline{outtea} \cdot vms$

Node 4 = $\overline{out1} \cdot cg \mid \overline{outtea} \cdot vms$

Node 5 = $\overline{out1} \cdot cg \mid vms$

Node 6 = $cg \mid \overline{outtea} \cdot vms$

Node 7 = $\overline{a} \cdot \overline{a} \cdot \overline{out1} \cdot cg \mid \overline{outtea} \cdot vms$

A useful extension to the program would be to have the growing derivation tree drawn in another window as each expression is displayed in the **Expansion** window. By making clear when a state

was reachable by different sequences of actions, such a display would provide further help to the program user in coping with the problem of combinatorial explosion that was referred to in Section 1. It would forestall the possibility that the user might unwittingly explore the same sequences of actions in different parts of a large expression for a system, while also giving a visual indication of which states in the system had and had not been investigated.

Indeed, the graphical output method would be so much more informative than the output in the `Expansion` window that program users would probably want to dispense altogether with the existing methods for selective expansion of sub-expressions, in favour of simply clicking in the graphical display window on the node which they wished to see expanded. In that case, the `Expansion` window would become merely an adjunct to the main interface to the program, of concern only to users with an interest in the textual language of CCS.

4 Evaluating CCS as a Specification Language

4.1 The Second Illustrative Specification

The specification we use as the basis for this discussion of features of CCS is that of an unbounded first-in, first-out queue. To avoid complicating the discussion, we specify just three operations on the queue: *enqueue*, *dequeue* and *length*. These operations are represented respectively by the actions $accept(X)$, where X, the item to be queued, is an input parameter, $\overline{return}(X)$, X an output parameter, and $\overline{length}(L)$, L being a natural number returned as output. In the specification, we model the empty queue by the atomic term *empty* and the non-empty queue as the structured term $q(X, Y)$, where X is the most recently arrived item, and Y is the queue of all items ahead of it.

The empty queue is capable of engaging in the two actions of accepting a new item or reporting the queue length to be 0:

$$queue(empty) \stackrel{\text{def}}{=} accept(X) \cdot queue(q(X, empty)) \; + \; \overline{length}(0) \cdot queue(empty)$$

After accepting an item, the empty queue behaves as defined in the expression for the non-empty queue, which may engage in three actions:

$$\begin{aligned} queue(q(X, Y)) \stackrel{\text{def}}{=} \; & accept(Z) \cdot queue(q(Z, q(X, Y))) + \\ & returnlength(q(X, Y), 0, empty) + \\ & returnhead(q(X, Y), empty) \end{aligned}$$

This agent expression does not define any actions for the second and third operations; it simply states that the actions will be those which occur in the defining expression for the agents *returnlength* and *returnhead*. Each of these has a definition for the empty and for the non-empty

queue:

$$\begin{aligned} returnlength(empty, L, Q) &\stackrel{\text{def}}{=} \overline{length}(L) \cdot reverse(Q, empty) \\ returnlength(q(X, Y), L, Q) &\stackrel{\text{def}}{=} returnlength(Y, L+1, q(X, Q)) \end{aligned}$$

$$\begin{aligned} returnhead(q(X, empty), Q) &\stackrel{\text{def}}{=} \overline{return}(X) \cdot reverse(Q, empty) \\ returnhead(q(X, q(Y, Z)), Q) &\stackrel{\text{def}}{=} returnhead(q(Y, Z), q(X, Q)) \end{aligned}$$

In *returnlength*, the second parameter, initially 0, is used as an accumulator while the queue is traversed from its tail to its head. When the empty queue is reached, the value of the accumulator is returned as parameter in the output action $\overline{length}(L)$. This is the only action in the pair of expressions that together define the agent. *returnhead* also has just one action in its two expressions, which is carried out when the front of the queue is reached, and X is found to be the item at the head of it.

The structure of each of these pairs of expressions implies a recursive traversal of the whole of the queue in order for the action which is the first, and only, prefix of each pair to be identified. That traversal destroys the representation of the queue in the first parameter, and the queue is reconstructed in the third parameter, beginning from the term *empty* as initial input parameter. The process reverses the order of the queue, so the agent *reverse*, which has no actions, is provided to reconstruct the queue in its original order:

$$\begin{aligned} reverse(empty, Q) &\stackrel{\text{def}}{=} queue(Q) \\ reverse(q(X, Y), Q) &\stackrel{\text{def}}{=} reverse(Y, q(X, Q)) \end{aligned}$$

We define a user of the queue as always being willing to engage in communication with the queue data structure via any of its actions, and the whole system to be a suitably renamed and restricted composition of the two components:

$$\begin{aligned} user(N) \stackrel{\text{def}}{=}\ & \overline{send}(N) \cdot \overline{enq}(N) \cdot user(N+1) + \\ & length(L) \cdot \overline{lengthIs}(L) \cdot user(N) + \\ & deq(X) \cdot \overline{receive}(X) \cdot user(N) \end{aligned}$$

$$system \stackrel{\text{def}}{=} (queue(empty)[e/accept, d/return] \mid user(5)[e/enq, d/deq]) \setminus \{e, length, d\}$$

The three operations on the queue are restricted to occur only as communication actions between the user and the queue. As communication events internal to the system, they will be visible to the observer only as undifferentiated *silent transitions* of the system. Each action of the user in communicating with the queue is accompanied by a second action that is not restricted: $\overline{send}(N)$, $\overline{lengthIs}(L)$ and $\overline{receive}(X)$. It is by noting the occurrences of these actions in the expressions for the system after applications of the expansion law, and in particular by noting the values taken by the parameters, that the observer will establish whether the specification of the queue data structure is correct.

The definition of the actions of the user process in invoking the enqueue operation ensures that successive invocations will cause succeeding integers, starting arbitrarily from 5, to be enqueued. A distinct value must be generated for each invocation for it to be possible to check whether the specification does capture the intended first-in, first-out behaviour.

4.2 Checking the Specification

The following user operations will be attempted on the queue:

Operation	Expected Result
$lengthIs(L)$	$L = 0$
$send(N)$	$N = 5$
$send(N)$	$N = 6$
$lengthIs(L)$	$L = 2$
$receive(X)$	$X = 5$
$receive(X)$	$X = 6$
$lengthIs(L)$	$L = 0$

Rather than simply presenting the result of selecting the path through the derivation tree that corresponds to this sequence, we show the expressions at some intermediate stages in order to illustrate issues involved in using the program and analysing its results.

The first application of the expansion law to the expression for *system* shows that two user actions are possible in the system's initial state:

```
T.(!lengthIs(0).user(5) | queue(empty)) +
!send(5).(!e(5).user(5+1) | (
    e(A).queue(q(A, empty)) +
    !length(0).queue(empty)))
```

The two actions are to engage in the internal communication event which is that of requesting the current length of the queue or to announce the $\overline{send}(5)$ action, as preliminary to enqueueing this value. The dequeue operation is not available to the user at this stage, as we intended when we wrote the specification. It should not be possible to invoke the dequeue operation when the queue is empty.

To carry out the first test, we select for expansion the sub-expression for the first of the two available operations. The program's output shows that the test yields the expected result:

```
T.!lengthIs(0).(user(5) | (
    e(A).queue(q(A, empty)) +
    !length(0).queue(empty)))
```

After the two enqueue operations, which were carried out by selecting the appropriate sub-expressions at each point when the user had a choice of operations, we were presented with the following program output, which confirms that the values 5 and 6 were indeed those enqueued. The three silent transitions represent communication between user and queue in carrying out the three operations on the data structure:

```
T.!lengthIs(0).!send(5).T.!send(6).T.(queue(q(6, q(5,empty))) | user(6+1))
```

After expanding this expresion, we find that all three operations on the queue are available as alternatives:

```
T.!lengthIs(0).!send(5).T.!send(6).T.(
   T.(!lengthIs(2).user(7) | reverse(q(5, q(6, empty)), empty)) +
   T.(!receive(5).user(7) | reverse(q(6, empty), empty)) +
   !send(7).(!e(7).user(7+1) | (
       e(A).queue(q(A, q(6, q(5, empty)))) +
       !length(2).reverse(q(5, q(6, empty)), empty) +
       !d(5).reverse(q(6, empty), empty))))
```

Here is the first serious difficulty: two of the operations are identified only by silent transitions. Which one of them is the request for the length operation that we wish to have carried out in order to complete the next test on the data structure? Needing to identify what action a particular silent transition represents is a recurrent problem in specification work with CCS. In the case of this example, the specification of the queue data structure can only be animated by bringing it into contact with a suitable environment, our *user*, yet as soon as one composes the test environment with the specification to be verified, that environment becomes part of the composed system, and its actions to exercise the specification are discernible only as silent transitions.

It was to circumvent this persistent problem that we defined the user process to carry out two actions for each possible operation on the data structure: one to communicate to the data structure the request for the operation and a second to report the fact of making the request. If this reporting action were always the first of the two, it would always be possible to distinguish alternatives in a complex expression, as each silent transition would occur after an externally visible action, and not after a branching of the derivation tree. However, in the present specification, where two of the operations on the data structure return to the user values which we wish to know, the request for the operation must come first. We might have defined the user expression with a third action for the dequeue and length operations, which, placed before the communication action with the data structure, would be a parameterless action announcing the imminent request for a particular operation. However, this seemed a cumbersome solution to a recurrent problem in CCS. As an alternative to it, one might modify the representation of silent transitions to make clear what communication event each represented. If such a scheme were adopted for the graphic display discussed in Section 3, it could be done without altering the syntax of CCS in its textual form.

In the above example, we could readily discern that the operation we required was represented as the first silent transition by studying the sub-expression that followed each. However, this will not always be so, as the display after the selection of this sub-expression illustrates:

```
T.!lengthIs(0).!send(5).T.!send(6).T.T.!lengthIs(2).(
   T.(reverse(q(5, q(6, empty)), empty) | !lengthIs(2).user(7)) +
   T.(reverse(q(6, empty), empty) | !receive(5).user(7)) +
   !send(7).(!e(7).user(7+1) | (
       e(A).queue(q(A, q(6, q(5, empty)))) +
```

```
        !length(2).reverse(q(5, q(6, empty)), empty) +
        !d(5).reverse(q(6, empty), empty))))
```

The display confirms that the specification of the length operation is correct in the case of the non-empty queue, and presents us with the familiar three alternatives for the next user operation. Unfortunately, they are not so familiar in this display, as the program has taken advantage of the commutativity of the composition operator to re-order composands in each alternative, and the user operations which give us the clue to the meaning of the silent transitions are now shown last. The program is also liable, when processing certain types of complex expression, to use the commutativity and associativity properties of the summation operator to simplify its computational task, and we cannot in all cases rely on the order of presentation of alternatives being preserved between one expansion and the next. Happily, no re-ordering of alternatives takes place in this example, and we may readily identify the sub-expression that corresponds to the dequeue operation, which is the next test case. After two invocations of it, the program display is:

```
T.!lengthIs(0).!send(5).T.!send(6).T.T.!lengthIs(2).T.!receive(5).T.!receive(6).(
    T.(queue(empty) | !lengthIs(0).user(7)) +
    !send(7).(!e(7).user(7+1) | (
        e(A).queue(q(A, empty)) +
        !length(0).queue(empty))))
```

The display indicates that values are returned in the order queued, so the specification of the enqueue and dequeue operations is correct. Finally, having observed that the dequeue operation is once more unavailable, we check that the queue length is again 0:

```
T.!lengthIs(0).!send(5).T.!send(6).T.T.!lengthIs(2).T.!receive(5).T.!receive(6).T.
                                                                !lengthIs(0).(
    T.(!lengthIs(0).user(7) | queue(empty)) +
    !send(7).(!e(7).user(7+1) | (
        e(A).queue(q(A, empty)) +
        !length(0).queue(empty))))
```

4.3 Distinctive Features of CCS

CCS is a formalism concerned with actions of processes and not with states of data structures whose values persist between actions. The difficulty of characterising in the notation operations which manipulate representations of stored data was the source of our problem in specifying the length and dequeue operations. The solution adopted, which involved adding extra parameters to some agents, devising a *reverse* agent and, in effect, treating agent definitions as expressions to be re-written, in the style of an OBJ-like algebraic language [Bur 77], was not especially hard to

fathom for this relatively simple specification, but the resulting set of definitions lacks the clarity in the statement of the meaning of operations that OBJ's equational semantics has.

The interesting insight that emerges from the investigation by means of the program is that the rather tortuous nature of the specification, and the potentially lengthy manipulation of terms as parameters of agents that occurs in seeking the prefix for expressions being transformed into standard concurrent form, is to a large extent concealed from the program user. As the commentary in Section 4.2 showed, the investigation is carried out by observing the sequence of actions as they emerge. Factors which influence the ease with which the sequence can be followed are the nature of the interface with the program and the manner in which internal communication is modelled in CCS. It is certainly not affected by any re-writing of expressions that the program may have to do to obtain expressions in standard concurrent form, all of which is completely hidden from the program user. Nor, once one has developed a reasonable facility for distinguishing in CCS expressions between prefixes that represent actions and summations and compositions that represent states, is the ease of the investigation compromised by the quite complex expressions that in Section 4.2 emerged to represent states of the queue data structure. As the emphasis of CCS is on process and not on state, so it is natural to focus on actions and not to pour over the details of the representation of a data structure in order to verify the correctness of the specification of an operation. One might out of interest study some of the examples in Section 4.2, and observe, for example, that:

```
queue(q(A, q(6, q(5, empty)))
```

represents the state of the queue that would result after the action `e(A)`, where `A` is a variable that denotes whatever value the user enqueues, but, unlike in the animation of an OBJ specification, one is not obliged to count brackets and check the order of values in a representation of a data structure in order to verify the operations.

5 Conclusion

There is much more work that could be done to make the program more helpful in the manner it presents the animation of a CCS specification. The most useful of the possible improvements that have been mentioned is probably the provision of a graphical interface to it, as discussed in Section 3.2. However, to the extent that the program palliates the disadvantages of CCS, while providing an interesting means for animating a CCS specification and thereby focussing attention on the notation's strengths, one may hope that it does represent a useful tool for users of the formalism.

References

[Bur 77] R. Burstall & J. Goguen. *Putting Theories together to Make Specifications* IJCAI 77, Invited Paper

[Mil 80] R. Milner. *A Calculus of Communicating Systems* LNCS 92, Springer Verlag, 1980

[Mil 89] R. Milner. *Communication and Concurrency* Prentice Hall, 1989