Modelling and Analysis of a Computer Conferencing System

Haig Baronikian, B.A.Sc., P.Eng.

Department of Electrical Engineering
McGill University
Montréal, Québec

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the
degree of Master of Engineering

October, 1987

Abstract

The main objective of this work is to develop a conceptual framework for the analysis and design of computer conferencing protocols, focussing on connection management and group-formation issues. A secondary objective is to gauge the utility of a particular model of parameterized, discrete communicating systems and its associated analytical techniques. The research describes three alternative formulations of increasing complexity. Formal descriptions are derived for *bridge* and *agent* processes which represent the distributed, interacting components of a conference system. These descriptions are analyzed and verified with automated tools. The architectural, behavioural and computational results for Single- and Two-Conference Systems are discussed. A high degree of modularity, generalizability and flexibility is attained in the process descriptions, allowing the formulations to cater to any number of conference participants or conferences and, a variety of connection and grouping schemes. At the same time, the model and techniques are found to be adaptable to the formulations and able to deliver expected results in a tractable, compact and efficient fashion.

Sommaire

L'objectif premier de ce mémoire est de développer un fondement conceptuel pour l'analyse et conception de protocoles pour la téléconférence informatisée, avec emphase sur l'administration des connexions et la formation de groupes. Ce mémoire a aussi comme deuxième objectif d'évaluer l'utilité d'un modèle paramètrique de système de communications discretes avec ses techniques d'analyse. La recherche décrit trois formulations alternatives de complexité croissante. Des descriptions formelles sont derivées pour les processus « ports » et « agents », qui representent les composantes interactives et réparties d'un système de téléconférence. Ces descriptions sont analysées et vérifiées au moyen d'outils mécanisés. Les résultats au niveau d'architecture, de comportement et de calcul pour des systèmes de téléconférence simples et doubles sont discutés. Un haut degré de modularité, de généralisation et de flexibilité est réalisé avec les descriptions de processus. Ceci permet d'étendre la formulation à un nombre arbitraire de conférences ou de participants à une conférence et à une varieté de possibilités de connexions et de formation de groupes. De plus, on a trouvé que le modèle peut être adapté à différentes formulations, et qu'avec les techniques associées il mène aux résultats attendus, d'une façon résoluble, compacte et efficace.

Acknowledgements

The author gratefully acknowledges the generous assistance of Dr. R. deB. Johnston during both the research and preparation phases of this thesis, and Messrs. G. Vonderweidt and T. Nguyen for many fruitful discussions.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

The need for *meetings-at-a-distance*, for reasons of time savings, financial economy and/or improved contact, is commonplace. Thus, there is growing practical interest in the construction of interactive (real-time, computer-assisted, multi-media) conferencing systems for business applications, certain educational settings, military uses, and in fact wherever there is a need for two or more agents (people or automata) to communicate concurrently towards a common goal. Such conferencing systems would supply the resources and operating framework, over either wide- or local-area networks, for agents to form groups and exchange voice, textual and/or pictorial information in a *(virtual) shared space*, a powerful paradigm which has been promoted by Thompson [1].

A number of trial, first-generation interactive conferencing systems, based on combinations of various technologies, are described in the literature [2] - [10]. Among the activities supported by these systems are group problem-solving and decision-making, computer-aided design, teletutoring, document editing, and collaborative

program development, all of which are forms of what Meyrowitz and van Dam term *distributed-knowledge work* [11].

1.1 Motivation and Goals

Experience suggests that there may be hidden difficulties in the design of complex communicating systems. In the case of interactive conference systems, such problems as consistency and synchronization of the database, treatment of acknowledgements, flow control, response time, and connection issues are mostly unexplored.

Little of a theoretical nature is available in the literature, with the exception of the important and extensive contribution of Sarin's work [12]. Sarin explores many architectural concerns, communication strategies, and performance issues. Two prototype systems are implemented and a third is proposed. Although communication plays a part in these prototypes, no attempt is made at formal analysis or logical verification of the methodology used. The emphasis, rather, is placed on timing and/or sequencing of exchanges between controller and conference participants. In another paper, Rea [13] studies information transfer in a small conference system with the objective of attaining mutual exclusion in the broadcasting of an uninterrupted series of data packets by either of two terminals. The management of both positive and negative acknowledgements, including filtering routines, for multicast systems is evaluated by Mockapetris [14]. Pardo and Liu present designs for several multidestination communication strategies for use in distributed systems [15]. Lastly, Birman and Joseph

[16] deal with the problems of reliable group communication under various modes of failure.

Perhaps as a result of, perhaps in spite of the lack of theoretical research, standard service descriptions, and design (realization) techniques for conferencing protocols, the need for conferencing systems has produced the implementations noted earlier, largely comprised of available products and services glued together on a relatively ad hoc basis. Therefore, it would be appropriate to formally model and analyze conferencing systems, with the objective of gaining insight into the behaviour, and thus, the design considerations of such systems. Recent attention by international standards organizations to the topic of conferencing and the underlying requirement for *multipoint/multicast services*, is further indicative of the timeliness of this project.

The term multipoint refers to the combined physical interconnection of more than two agents, while multicasting denotes the logical or functional relationship held between agents and is based on addressing facilities. In the present work, the basic multipoint/multicast connectivity problem is undertaken with the following goals:

- To develop a conceptual framework and a notation, in terms of distributed processes, for describing the connectivity of agents in conference groups;
- To generalize the above results to handle conferencing systems of arbitrary size, while exploring the application of parameterization; and,
- To ascertain the effectiveness of a set of automated tools in studying and verifying the process descriptions generated from the above two steps.

Certain limitations will be placed on the scope of these goals once some prelimi-

nary ideas are dealt with in the following section.

1.2 Basic Concepts

The study of most systems, whether for analysis or synthesis, begins with the decomposition of that system into a set of interacting subsystems or processes. Thus, for simplicity, the conceptual conference model chosen consists of two processes (see Figure 1.1): an *intelligent bridge* or meeting room that delivers virtual N-way connections and is capable of performing predicate-based atomic tests, and, an arbitrary number of *conference agents* or participants. The agents may connect together in star configurations to form *conference groups* or meetings, by means of the bridge. Interaction or communication between the bridge process and the agent processes, as well as between the composite system of the two and the environment, is achieved through discrete *messages* or statements. The syntax, semantics and relationships governing these messages define a *protocol* or rules of dialogue. In effect, this provides the basis for multipoint/multicast connection services for users, and includes point-to-point connections as a subsidiary function.

A single bridge process could handle several multipoint/multicast *conferences* at once. Within each conference, *channels* may exist to cater to differently characterized data-streams. For example, a stream for control and voice, a second for real-time shared-space interactions between agents, and a third for slower bulk file transfers, would be suitable for many applications. Such an operation could be accommodated on the 2B+D channelized subscriber access vehicle of the Integrated Services Digital

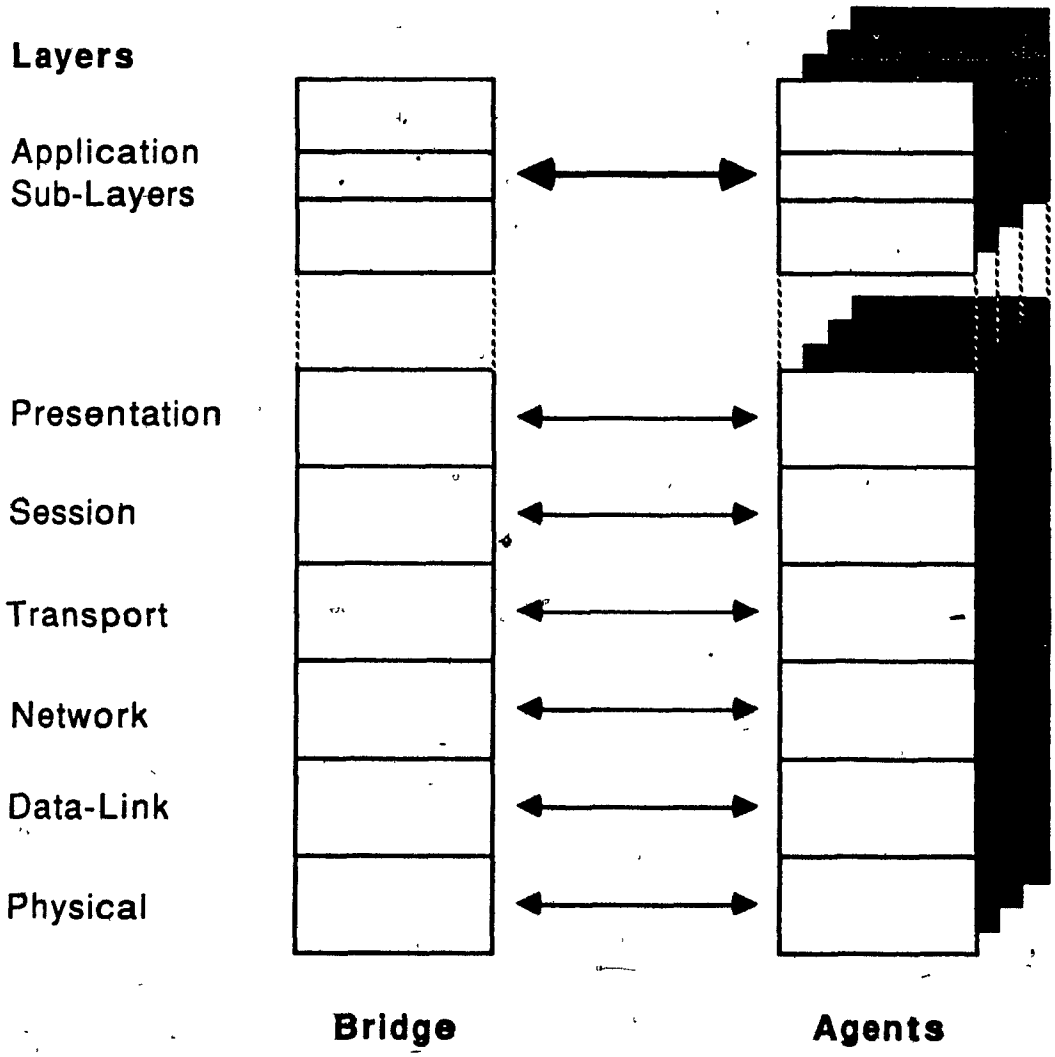


Fig. 1.1 ISO Layered Architecture for Open System Interconnection

Network (ISDN) [17]. In fact, shared-space conferencing has already been suggested as an application for ISDN [18].

A variety of connection or call establishment scenarios are possible, depending on the initiator of the connection. Most commonly perhaps, an agent identifies itself and initiates a conference by receiving, manipulating, and passing on an appropriate *service_request* message from its environment to the bridge. Once the request is accepted, subsequent requests are made to the bridge for the inclusion of other *target* agents in that conference. This procedure is termed DIAL-OUT. Other scenarios are also possible including MEET-ME, where an agent seeks permission and joins an existing conference.

As stated earlier, very little has as yet appeared in the way of international standards for conference systems (interactive or not). The relationship of a protocol which provides basic multipoint/multicast services, as well as more advanced conference management features, to the Open Systems Interconnection Basic Reference Model is unclear [19]. Although the layered architecture of the Reference Model, displayed in Figure 1.1, has allowed for multipoint/multicast facilities in the form of *decentralized multi-endpoint connections*, there is some confusion over details for its use [20]. It is most likely that, given the Reference Model is rather well-defined for point-to-point connections, the basic services and more advanced conferencing features would fit as sub-layers of Layer 7 (Application).

For the present research, only the bridge-agent protocol, and a set of *basic*, primary services are investigated, leaving aside bridge-bridge communication for future study. A number of further limitations are imposed, all of which help contain the complexity of the task at hand:

- no probes of or acknowledgements to target agents are implemented;
- the initiator of a conference, the *chairman*, controls that conference until its clearing;
- disjoint conference groups are used, with unique names for agents and conferences;
- information transfer is not modelled; and
- only a single channel is implemented for each conference.

1.3 Outline

Having briefly delved into the background for interactive conference systems and considered the motivation for the current work, the following chapter surveys formal description techniques and details the mathematical foundations and analytical tools which will be utilized for the ensuing venture.

Chapters 3, 4, and 5, describe three successively more intricate and flexible formulations for a conferencing system. All solicit some degree of parameterization. The respective results of the application of automated tools to these formulations are also presented.

Finally, Chapter 6 summarizes the findings, states the general conclusions and provides some remarks regarding possible future extensions of this work.

Chapter 2 Modelling Approach

The present chapter is mainly devoted to familiarizing the reader with the various formal description techniques (FDTs) used in the modelling and analysis of distributed or communicating processes. Such formalisms attempt to overcome the ambiguities and incompleteness of informal methods (e.g. natural language specification) and, often lead to compact and mechanizable descriptions. Thus, it is not surprising that many FDTs have found application in the description of communication protocols, since protocols may be regarded as processes.

The chapter begins with a general discussion and categorization of FDTs. The particular FDT used for the current task of studying conferencing systems, originating from an algebraic model of Discrete Communicating Processes (DCP), is then introduced. Lastly, a set of automated tools based on DCP are outlined.

2.1 Overview of Formal Description Techniques

There are two major approaches to the formal description of processes, with a

third being a hybrid of the first two. Sunshine [21] provides a table of the various techniques and their characteristics. An alternative classification, grounded on state complexity (the encoding of information within a state) and language expressive power (the encoding of requirements on the sequence or history of states), is given by Schwartz and Melliar-Smith [22].

The first approach arises from the basic paradigm of a *finite-state machine* (FSM), consisting of a finite set of states and the available *transitions* to travel between those states. Transitions in one process are imagined to occur instantaneously, and may be linked to the transitions of another process. Danthine [23] represents the FSM as a 5-tuple, with a finite set of states, inputs and outputs, and state-transition and output functions. Processes modelled as FSM's have a graphical equivalence and may be examined through reachability analysis, the enumeration of the reachable states in the Cartesian product state-space of the processes.

FSM-based techniques appear to model well the control aspects of processes, but are completely inadequate, by definition, for (possibly) infinite-state processes. A secondary complaint is the phenomenon of state explosion, which is experienced for any but the simplest situations. One method of alleviating this difficulty has been the use of context variables and procedures in order to reduce the set of states, but doing this frequently makes the process less tractable from the designer's viewpoint. An example of an end-to-end protocol modelled as FSM's may be found in [23]. Another specimen of a communications protocol is available in [24].

A model closely related to the FSM is the *Petri Net*. The Petri Net is appealing because of its facility for visual depiction of processes, but also lends itself to compact matrix representation. The ordinary Petri Net is a graph with places, transitions, directed arcs, and tokens. In Danthine's terminology [23], this is a 4-tuple, where the sets of places and transitions simulate conditions and events respectively, and input and output functions map conditions to events and vice versa. Beginning with an initial marking of a graph, a transition is enabled when there is at least one token at each of the transition's input places. When a transition fires, one token is removed from each input place and one is added to each output place. The state of the system is given by the distribution of tokens around the graph.

Petri Nets bring control structures to the foreground, and may be made capable of modelling infinite-state systems by increasing the number of tokens in a net without bound [25]. However, they are still prone to the annoyance of state explosion. Many extensions have been developed to the ordinary Petri Net, including the Coloured-Arc [26] and Time [23] flavours. Danthine [23] gives the same end-to-end protocol mentioned earlier in terms of the Petri Net. An additional sample of a communication protocol modelled as a Petri Net is given in the work of Diaz [27].

The second major approach to formal description is borne of programming paradigms and high-level languages. Here, a process is described as a set of assertions or an algorithm. The analysis is performed using assertion proofs. This approach is usually better suited to the data transfer aspects of processes, but application to

connection-oriented problems is relatively difficult [21]. Since a high-level language may be designed to be universal in scope, it can extend much further than either FSM's or Petri Nets in modelling power. Nevertheless, the ability to do so is limited by the difficulty of finding and verifying assertions. Utility is also hampered by the elusiveness of automation for proof techniques, in comparison to the rather direct reachability analysis of FSM's and Petri Nets. Tanenbaum [28] offers examples of communication procedures for the data-link layer of OSI, including some sliding-window protocols, expressed in the PASCAL programming language.

As it has become obvious that each of the above approaches has its own advantages and drawbacks, many researchers have opted for a hybrid approach, combining the best elements of each model and accompanying analytical mechanism. A working group of the International Standards Organization has been studying a hybrid FDT called Estelle, which has strong similarities to the PASCAL language [29]. Estelle uses a FSM model augmented with context variables, one of which is the special STATE variable, and predicates on those variables. The application of Estelle is illustrated with the Alternating-Bit Protocol. Further proposals of hybrid methodologies are those of [30], [31] and [32]. Some investigations of communications protocols, exploiting *parameterized* hybrid techniques, are covered in [33] and [34].

2.2 The Discrete Communicating Processes Technique

The modelling and analytical technique used for the ensuing research is founded on the Discrete Communicating Processes (DCP) construct of Johnston [35]. DCP

essentially models a process as a FSM which interacts with its environment via *discrete*, instantaneous messages. Furthermore, a process is treated as a *black box*, where internal details are too complex, not of interest or unavailable.

In order to study processes, including communications protocols, it is generally necessary to have a means of differentiating and comparing processes. The behavioural or black box approach of DCP readily lends itself to the task by characterizing processes according to their *externally visible* behaviour. Thus, when two given processes exhibit the same pattern of message interchange with their respective environments, as viewed by an external observer, the processes are considered equivalent.

In this context, a process evolves by offering to exchange only one of a fixed set of messages with its environment (which may include other processes). The model assumes that the source and the destination of the message are *synchronized* such that both await the occurrence of the *communication event*, and then progress simultaneously with transitions to new states or *successor processes*. As each state may offer a different pattern of behaviour, states may be thought of as distinct processes. Hence, notions of process and state become intermingled.

DCP shares all of the above primitive notions with its close relative, the Calculus of Communicating Systems of Milner [36]. DCP, CCS and Hoare's Communicating Sequential Processes [37] allow for the modelling of concurrent processes, and

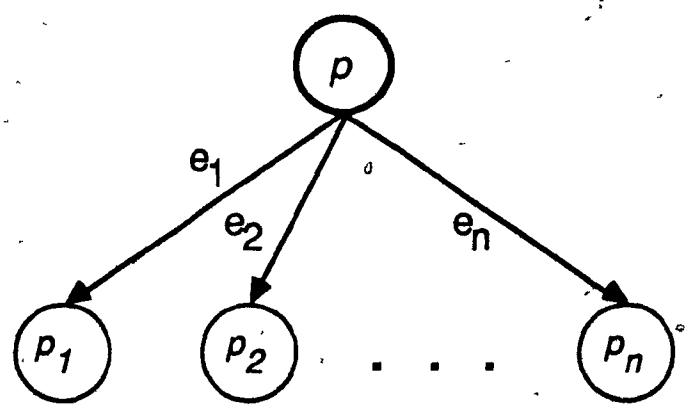
with value-passing, infinite-state systems. Of the three, DCP is highly amenable to automation.

In DCP, a process p is defined as a set of ordered pairs, < communication event, successor process >, where processes are expressed in terms of other processes, as follows:

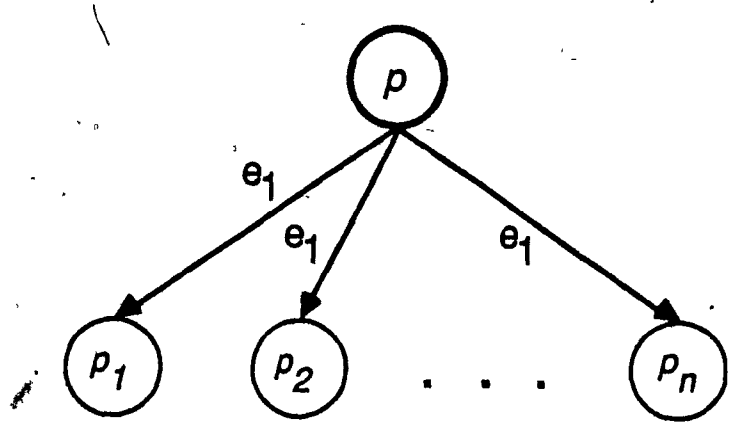
$$p = \{ < e_1, p_1 >, ..., < e_n, p_n > \}$$

A tree-based graphical interpretation is given in Figure 2.1(a), where the node p is a process, the directed arcs $e_1, ..., e_n$ are communication events, and the nodes $p_1, ..., p_n$ are the respective successor processes. Alternatively, algebraic, matrix or state-transition diagram representations may be utilized. It is noteworthy that the graph of a finite-state process would have a finite set of nodes.

Some processes are fully known or predictable and are said to have *deterministic* behaviour. Figure 2.2 gives an example of a deterministic Client process, which requests a resource unit, and upon receipt of that unit, continues by making a request for a second, indistinguishable unit. Once it holds two units, it returns them one at a time to reach its initial (empty) state. In contrast, many processes are either difficult to model or are inherently unpredictable. These are said to be *non-deterministic*. A case in point is the Client process illustrated in Figure 2.3, which has the random behaviour of returning the first unit, or developing as before to request the second unit. Whereas a deterministic process has one successor process for each communication



(a) A Deterministic Process



(b) A Non-Deterministic Process

Fig. 2.1 Graphical Interpretation of Processes

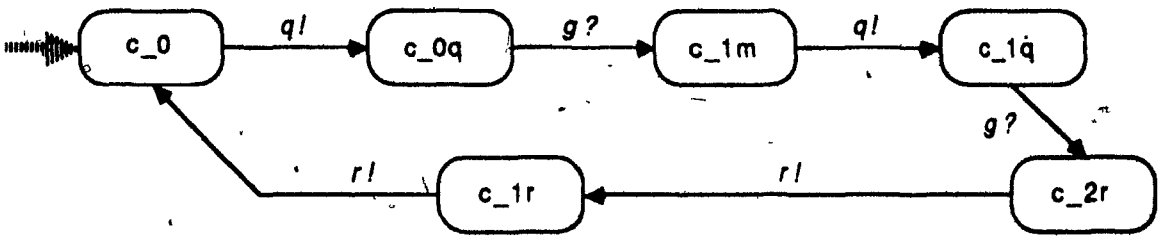


Fig. 2.2 A Deterministic Client Process

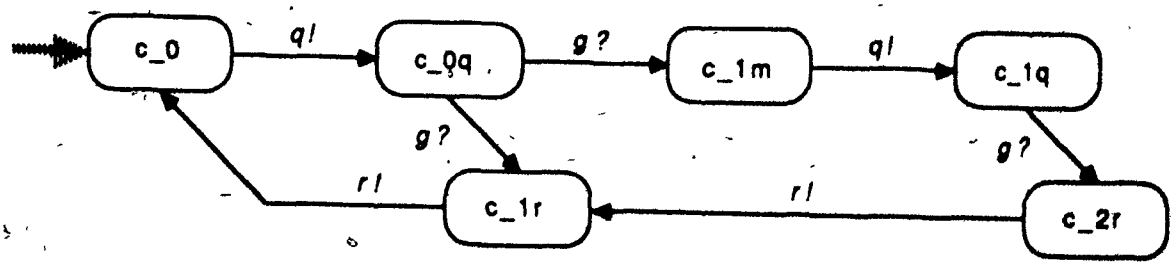


Fig. 2.3 A Non-Deterministic Client Process

event (Figure 2.1(a)), a non-deterministic process has several successor processes for a given communication event (Figure 2.1(b)). The two Client processes of Figures 2.2 and 2.3 are stated below, in recursive fashion, with the extra transition due to the non-determinism shown in bold (following Hoare's notation [37], ! indicates an output event and ? indicates an input event):

c_0 = { < q !, c_0q > }

c_0q = { < g ?, c_1m >, **< g ?, c_1r >** }

c_1m = { < q !, c_1q > }

c_1r = { < r !, c_0 > }

c_1q = { < g ?, c_2r > }

c_2r = { < r !, c_1r > }

Thus, in the non-deterministic situation of Figure 2.3, the occurrence of the communication event g ? allows the random selection of a successor process from the two available.

The consideration of the relationship between these two example systems leads to the desire, cited earlier, for comparison of processes and indeed to the idea of a process ordering mechanism. For this purpose, the partial ordering relation $\leq$ is identified. In the two examples of the Client processes, clearly the second is able to behave as the first; but, the converse is not true. That is, while both processes are able to request and return up to two resource units, one at a time, only the non-deterministic Client process can request and return a single resource unit without continuing the

communication for a second resource unit. If the first process was named q and the second p, the relationship between the two could be written as:

$$p \leq q$$

Again a graphical interpretation is possible for this relation, q being a sub-tree or sub-diagram of p, with correspondences extending over the communication events (process names may be different). The relation also brings forth the ideas of *simulation*, p mimics all the actions of q, and *containment*, p contains the sub tree or sub-diagram of q.

If the relation $q \leq p$ had also held, then, since the $\leq$ relation is reflexive, transitive and antisymmetric, an equivalence would be implied, as follows:

$$p \leq q \wedge q \leq p \Rightarrow p = q$$

The equivalence relation creates a partition of equivalence classes on processes. Two processes are then considered equivalent if they belong to the same equivalence class, and thus each offers precisely the same behavioural possibilities. A full and excellent exposition on equivalence classes may be found in [38].

Finally, the extension of these concepts to *interconnection* would be advantageous. The goal of interconnection in systems design is to predict the behaviour of a composite system formed of individual processes. Such a function must unify messages to form communication events (or link communication ports), and then hide these events or ports from the external observer. In DCP, as in Milner's CCS, this is

accomplished in two steps, using two distinct operators in order to retain the property of associativity. The subject is touched upon in greater detail in Subsection 2.2.2.

2.2.1 Mathematical Foundations

The first stage in establishing the mathematical foundations for DCP is the formulation of a process space. This is accomplished with the aid of the theories of partial orders, lattices and fixpoint semantics. The development is detailed in [35], and follows the outlines of [13] and [39].

Letting a set P_E denote the set of all processes that exchange messages from a finite set E, and accepting that any $p \in P_E$ is a set of ordered pairs, < communication event, successor process > as before, then p must be a subset of the Cartesian product $E \times P_E$. This may be expressed as,

$$p \in \mathcal{P}(E \times P_E),$$

where $\mathcal{P}$ is the power-set operator. As this applies for any $p \in P_E$, then,

$$P_E \subseteq \mathcal{P}(E \times P_E).$$

Any element of $\mathcal{P}(E \times P_E)$, being a set of event-process pairs, could also be imagined to be a process, and hence:

$$\mathcal{P}(E \times P_E) \subseteq P_E,$$

leading to the assertion,

$$P_E \subseteq \mathcal{P}(E \times P_E) \wedge \mathcal{P}(E \times P_E) \subseteq P_E \Rightarrow P_E = \mathcal{P}(E \times P_E).$$

For reasons of cardinality, however, the claim cannot be satisfied by any P_E, according to Cantor's Theorem [40]. This difficulty is abridged by redefining the power-set operator not to include all possible subsets. Instead, the set $\mathcal{P}(E \times P_E)$ is partitioned into equivalence classes with the use of partial ordering, producing a quotient set. Now, the space P_E is obtained by progressively building, for all $i \geq 0$, the intermediate finite process spaces $P_E^{(i)}$, each of which is a partial order with ordering relation $\leq_i$ and elements which are trees of at most length i. These serve as a series of approximating spaces whose limit, interpreted appropriately, is the desired space. Each space $P_E^{(i+1)}$ is defined recursively from $P_E^{(i)}$, beginning with the first space $P_E^{(0)}$. $P_E^{(0)}$ consists of a single member, a tree of length 0, representing a completely undefined and unconstrained process. Thus, it is possible to state that,

$$P_E^{(i+1)} = \lfloor P_i \rfloor_{\simeq},$$

where

$$P_i = \mathcal{P}(E \times P_E^{(i)})_{\perp}, \quad \text{for } i \geq 0,$$

and the elements of the power-set P_i are the sets of ordered event-process pairs, augmented with the bottom element $\perp$. The *limiting* space P_E as $i \to \infty$ is also a partial order with a bottom element $\perp$ and a top element $\top$ (the null process, which is one that offers to exchange no messages with the environment). The relation $\leq$ between classes $\lfloor P_i \rfloor$ is actually defined in terms of the relationship $\sqsubseteq$ between elements (representatives) of the classes. $\sqsubseteq$ is a pre- or weak ordering (reflexive and transitive), which induces the quotient partial order set, $\lfloor P_i \rfloor_{\simeq}$. Therefore, $\lfloor P_1 \rfloor \leq \lfloor P_2 \rfloor$ iff $P_1 \sqsubseteq P_2$. P_E is said to be a *complete* partial order since any directed

set D of elements in P_E has a least upper bound (lub), which is also contained in P_E. A directed set is one where any two elements have a lub in the same set.

There are two fundamental functions defined over P_E. The first is the pair-formation function $e; : P_E \rightarrow P_E$, where $e \in E$, serving to prefix a process p with the event e. That is:

$$e ; p = \{ < e, p > \} .$$

The second one is the alternation function $+ : P_E \times P_E \rightarrow P_E$, which aggregates behaviours and may be interpreted as set union. For example:

$$p + q = p \bigcup q ,$$

is a process containing the behaviours of either p or q. This is the greatest lower bound (glb) of p and q.

Both the pair-formation and alternation functions are monotonic and continuous, preserving ordering and least upper bounds. A number of useful properties, due to the partial order nature of P_E, may now be identified. Although proofs are omitted for brevity, the interested reader is directed to [35] for elaboration. The properties are:

- $e_1 ; q_1 \leq e_2 ; q_2 \Leftrightarrow ((e_1 = e_2) \wedge (q_1 \leq q_2))$.
- $x_1 + \ldots + x_N \leq e ; q \Leftrightarrow x_i \leq e ; q$ for some i.
- The glb of any set of processes exists. For $\{ p, q \}$, where $p, q \in P_E$, it is given by $p + q$.
- $\forall p \in P_E, p = \sum_i e_i ; q_i$, where the summation occurs over all $e ; q \geq p$.

A process $p \in P_E$ is termed finitely-branching if it can be denoted as a finite sum of elements, $e; q \in P_E$, or:

$$p = \sum_{i=1}^{N} e_i ; q_i \quad \text{. for finite N.}$$

Finally, it may be shown that finite-state processes can be uniquely or canonically specified as fixpoints of continuous, multi-dimensional functions which are directly built from the preceding functions. The rationale derives from the Fixpoint Theory of Program Semantics for complete partial orders [41], and the availability of monotonic and continuous functions defined upon them of the form $F : (P_E)^n \cdot (P_E)^n$.

2.2.2 Process Operators

Operators or functions are required to be monotonic and continuous in order to guarantee the existence of fixpoints. An intuitively appealing function to begin with is process interconnection, which permits the formation of larger systems from component processes. The * operator is used for this purpose, being defined for the moment as:

$$p * q = R (p \,||\, q).$$

It may be seen that the * operator can be indirectly obtained via the || and R operators. The first of these, the ||-product or asynchronous composition operator, is defined briefly here as:

$$p \,||\, q = \sum_i e_i ; (p_i \,||\, q) + \sum_j e_j ; (p \,||\, q_j) + \sum_{corr: k,l} - ; (p_k \,||\, q_l) .$$

In this recursive expression, the terms $\sum_i e_i$; (p_i [] q) represent the sum of possible events, e_i post-fixed with ! or ?, which can be exchanged between process p and the external environment, with the []-product resuming as p_i [] q. The terms $\sum_j e_j$; (p [] q_j) indicate the analogous case for q. The latter terms $\sum_{coev\ k,l}$ − ; (p_k [] q_l) denote an internal exchange or trace event between processes p and q, from which the []-product resumes as p_k [] q_l. The trace events are those events which appear in the event sets of both processes. In other words, these are event pairings which are identical in name (the name is retained for ease of identification) and complementary, with one event having the ! post-fix and the other the ? post-fix. All other events are present in only one event set and are presumed to be directed to or from the environment. Graphically, solid lines are used to represent external events, and broken lines represent trace events. Further details of this definition may be found in [35]; Milner's |-operator is almost identical [36].

The practical effect of the []-product is exemplified in Figure 2.4, which shows the earlier non-deterministic Client process communicating with a Timer process. The Timer may have the responsibility here of informing the Client process, who may prefer to obtain resource units on particular days, that it is Wednesday (for instance). If it is supposed that the two processes have unidentified, but complementary communication events or traces to cater to the task, the recursive equations for *Client* [] *Timer* may be written in the ordered-pair notation as:

```
c_0  []  t_0  =  { < -  , c_0'  []  t_0 > }

c_0'  []  t_0  =  { < q !, c_0q  []  t_0 >, < WED !, c_0'  []  t_0 > }
```

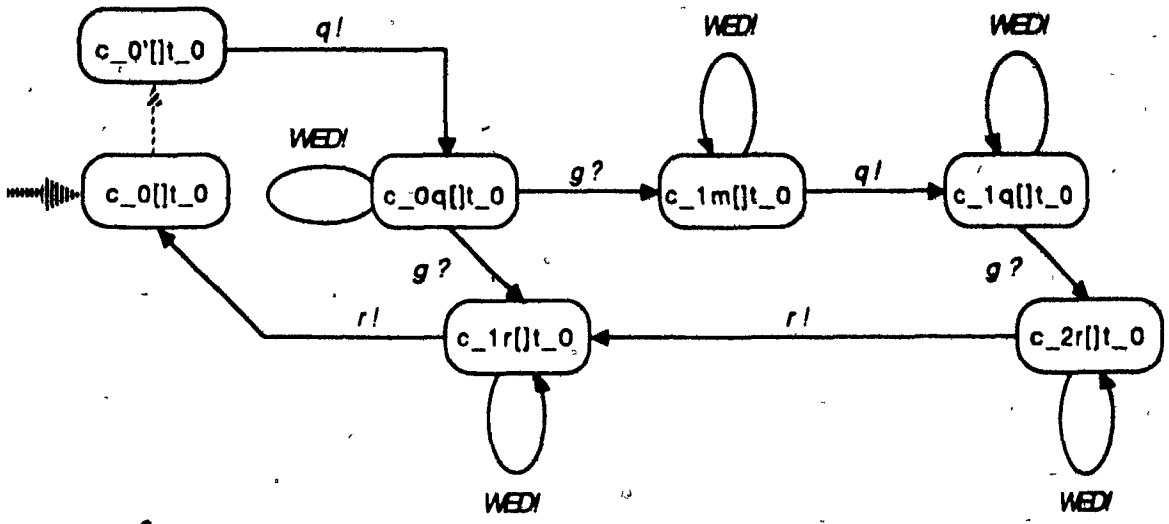


Fig. 2.4 []-Product of Non-Deterministic Client Process with Timer Process

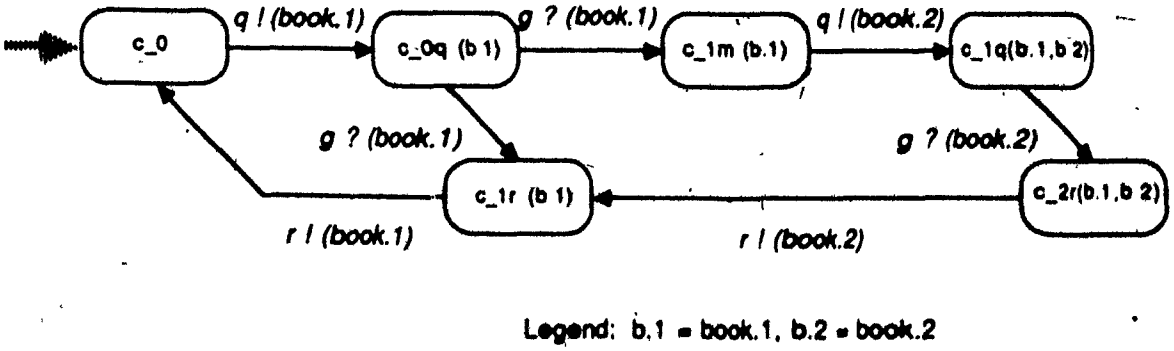


Fig. 2.5 Parameterized Non-Deterministic Client Process

```
c_0q [] t_0 = { < g ?, c_1m [] t_0 >,   g ?, c_1r [] t_0 >,
                                        < WED !, c_0q [] t_0 > }

c_1m [] t_0 = { < q !, c_1q [] t_0 >, < WED !, c_1m [] t_0 > }

c_1r [] t_0 = { < r !, c_0  [] t_0 >, < WED !, c_1r [] t_0 > }

c_1q [] t_0 = { < g ?, c_2r [] t_0 >, < WED !, c_1q [] t_0 > }

c_2r [] t_0 = { < r !, c_1r [] t_0 >, < WED !, c_2r [] t_0 > }
```

where, c_0 = {< q !, c_0q >} of the Client process is displaced by c_0 = {< WED !, c_0' >} and c_0' = {< q !, c_0q >}, and the Timer process is defined as t_0 = {< WED !, t_0 >}.

Thus, with the []-product, it is possible to follow both the external and internal (trace) transitions of an interconnected system. For the process interconnection function, it would also be beneficial to be able to divorce the externally visible behaviour from the traces, thereby *hiding* the latter. A second operator is defined to this end.

The R operator, which has some similarity to Milner's Restriction operator (in that internal events are suppressed), defines the potential external behaviour of a system. This concept is best explained through an example. If the system of Figure 2.4 were to be viewed externally, the trace event would be hidden, and the composite process would be similar to the simple non-deterministic process of Figure 2.3 but for *spontaneous* emissions of WED ! (which could occur if the process was to take more than a week in real time to complete). The ordered-pair expressions now take the

following form:

```
R (c_0  [] t_0) = R (c_0' [] t_0)

R (c_0' [] t_0) = { < q !, R (c 0q [] t 0) >,
                    < WED !, R (c 0' [] t_0) > }

R (c_0q [] t_0) = { < g ?, R (c_1m [] t_0) >,
                    < g ?, R (c 1r [] t_0) >,
                    < WED !, R (c_0q [] t_0) > }

R (c_1m [] t_0) = { < q !, R (c 1q [] t_0) >,
                    < WED !, R (c_1m [] t_0) > }

R (c_1r [] t_0) = { < r !, R (c_0  [] t_0) >,
                    < WED !, R (c_1r [] t 0) > }

R (c_1q [] t_0) = { < g ?, R (c 2r [] t 0) >,
                    < WED !, R (c 1q [] t_0) > }

R (c_2r [] t_0) = { < r !, R (c_1r [] t_0) >,
                    < WED !, R (c_2r [] t_0) > }
```

Formally, the R-operator is specified as:

$$R(p) = \sum_{<e,q> \in p} \tau(e,q) \text{ , where,}$$

$$\tau(e,q) = \begin{cases} e \ ; \ R(q) & \text{if } e \neq \\ R(q) & \text{if } e = \end{cases}$$

That is, when an event is externally visible, the event is kept and the R is imposed on its successor process. When the event is a trace, the trace is removed, leaving the R of the successor process.

One last matter must be addressed for the interconnection of processes to succeed: the question of *stability*. In the present model, processes are said to be stable if, for every pair of internally-connected processes or states, p_i and p_j, it is true that:

$$R(p_i) = R(p_j).$$

When this is indeed the case, the original statement, $p * q = R(p [] q)$, remains correct. For a stable process, proper classification of states into equivalence classes, hence extraction of the aptly simplified, externally-visible behaviour of that process is guaranteed. If a process does not meet the above stability criterion, a different operator is required to treat unstable transitions appropriately. The V operator (see [35]) serves that purpose, and displaces the R such that, for unstable systems:

$$p * q = V(p [] q).$$

As the implementation of the V operator was not available at time of research, analysis was carried out with the R operator only. This required very careful interpretation of the interconnection results.

Other operators include *co*, *det*, etc. These are not discussed here, but may be perused in [35].

2.2.3 Parameterization

As will be seen in succeeding chapters, parameterization plays an important role in the description of processes. It serves both to maintain compactness in descriptions, and to facilitate information-passing. Both of these concepts are intimately related.

All of the previously described operators function analogously for a parameterized discrete-event process as they did for the non-parameterized (instantiated) case [42]. For example, consider Figure 2.5 which is similar to the non-deterministic Client process of Figure 2.3. Behaviourally, the process is identical. However, messages are now defined with the parameter *book*.1 or *book*.2 ranging over the set *book*(s). The unparameterized process did not account for the identities of the resource units. They were, in fact, indistinguishable from each other. If the Client is now considered to be borrowing from a library, the library certainly cares about which books are on loan, and must distinguish them on some basis, perhaps the title. In turn, the Client must also keep track of what has been borrowed so that the correct book(s) is returned. This feat is delegated to the parameters, which appear in both messages, via which specific books are requested, granted and returned, and in states, where the titles of books are remembered between actions and trips to the library. Algebraically, this process is written as:

```
c_0  = { < q !  (book.1), c_0q (book.1) > } | book.1 ∈ book

c_0q (book.1) = { < g ?  (book.1), c_1m (book.1) >,
                  < g ?  (book.1), c_1r (book.1) > }

c_1m (book.1) = { < q !  (book.2), c_1q (book.1,book.2) > }
                                          | book.2 ∈ book

c_1r (book.1) = { < r !  (book.1), c_0 > }

c_1q (book.1,book.2) = { < g ?  (book.2), c_2r (book.1,book.2) > }

c_2r (book.1,book.2) = { < r !  (book.2), c_1r (book.1) > }
```

For the []-product, matches between a pair of communicating processes must now be formed on the basis of identical parameter (set) names, parameter list ordering, and parameter\list cardinality, in addition to the previous matching on the basis of event names. This is a more extensive, *syntactic* matching procedure.

2.3 Automated DCP-Based Tools

DCP-based automated, symbolic computation tools, implemented in VAX LISP (essentially COMMON LISP) are utilized to perform the tasks of analysis and logical verification of conferencing systems. The operating environment is the VAX 8600 computer running under VMS.

2.3.1 Defining Processes

The generalized format of a parameterized transition in the DCP process description is:

$$state\_name\ [global\_bindings] \ =$$

$$\sum\ +\ \#\ [local\_bindings]\ .\ message\_name\ !\ [parameters]\ ;$$

$$successor\_state\_name\ [parameters]\ -->\ (n)$$

The parameters present in the *message_name* and *successor_state_name* are bound to (i.e. parameter values are obtained from) the *global_bindings* and *local_bindings* which are themselves lists of variables or parameters. This allows for inter-state as well as inter-process information-passing, simulating an ordered read/write

memory for the processes. A + normally precedes every transition, and depicts non-determinism when two or more successor processes are available for the same *message_name*, except in certain instances as will be noted. The # implies the selection of a parameter value from a set. As an example, # (*x*.j) expresses the assignment of a value from the set *X* to the j_th parameter, *x*.j, based on that set. By convention, ! indicates an output message and ?, in the same position, indicates an input message; the absence of either indicates an internal event in a composite system. *n* is the index of the successor state in the state-transition table.

The means for defining a process to the automated tools is the process description file. The LISP *setq* function is applied to create an atom named after the process, whose symbolic value is the underlying recursive series of processes, each consisting of a set of parameterized < communication event, successor process > pairs. With reference to the generalized transition format, the parameterized Client process of Figure 2.5 may be written in equation form as:

```
c_0  =  q !  (book.1) ; c_0q (book.1)

c_0q (book.1) =  g ?  (book.1) ; c_1m (book.1)
               + g ?  (book.1) ; c_1r (book.1)

c_1m (book.1) =  q !  (book.2) ; c_1q (book.1, book.2)

c_1r (book.1) =  r !  (book.1) ; c_0

c_1q (book.1, book.2) =  g ?  (book.2) ; c_2r (book.1, book.2)

c_2r (book.1, book.2) =  r !  (book.2) ; c_1r (book.1)
```

and would thus be defined by the following *setq* assignment:

```
(setq client '(
                ((c_0)
                 ((#book.1) (q !     #book.1)
                            (c_0q #book.1)))

                ((c_0q #book.1)
                 (( )       (g ?     #book.1)
                            (c_1m #book.1))

                 (( )       (g ?     #book.1)
                            (c_1r #book.1)))

                ((c_1m #book.1)
                 ((#book.2) (q !     #book.2)
                            (c_1q #book.1 #book.2)))

                ((c_1r #book.1)
                 (( )       (r !     #book.1)
                            (c_0)))

                ((c_1q #book.1 #book.2)
                 (( )       (g ?     #book.2)
                            (c_2r #book.1 #book.2)))

                ((c_2r #book.1 #book.2)
                 (( )       (r !     #book.2)
                            (c_1r #book.1)))                ))
```

All of the above *state_names*, *message_names*, *successor_state_names* and *parameters* are literals which would normally be enclosed in double quotes. For clarity, the quotes have been omitted.

2.3.2 Key Modules

The automated DCP tools embody several main program modules, including some diagnostic utilities [42]. One module executes various checks on process description

files and prepares the files for subsequent manipulation by establishing internal data structures. The checks are based on syntactical verification on individual transitions and on proper recursion of the entire process, usually uncovering missing or misnamed *state_names*. Some errors are also flagged by the LISP interpreter upon loading of a process description file. As a last feature, the module has *pretty-printing* procedures which translate the process descriptions from their input form into a more appealing display.

A second module contains the basic DCP operators mentioned previously, the ||, R, *co*, and *det*. Here, unification and ordering algorithms are the major components. These algorithms are employed chiefly in the ||-product operator to supply syntactic matching in order to seperate externally visible behaviour from trace events. Forced matching is attempted when the algorithms note a *close* match. The two messages, *my_message*! ($x.1$, $y.1$, $z.1$) and *my message* ? ($x.2$, $y.1$, $z.1$) offered by two processes are almost complementary. The index of the first parameter of each message, however, is different, to resolve this, a match or unification would be ventured with the assumption, $x.1 = x.2$.

A third module supports the * operator (which invokes the ||, R and appropriate simplification routines), as well as the relations < and ' Two other modules are under development.

Chapter 3 A Preliminary System

The discussion of formulations and experimental results commences with the Preliminary System (PS) which serves as an aid to conceptualization, and a means of verifying correct operation of the formulated processes before continuing on with more complex systems. In this work, there have in fact been several *preliminary* systems, beginning with a completely non-parameterized, but unwieldy, formulation. Two studies of a semi-parameterized system are reported. The first inquires into a single-conference system consisting of a bridge and one agent, and the second, into a two-conference system embracing an *extended* bridge and two agents. Section 3.1 prefaces the studies by describing the elements common to both systems and, working inward from the environment, the agent and bridge processes.

3.1 Formulation of Processes

The basic purpose of the conference bridge in both studies is to accept requests for service and whenever possible, to grant those requests. The purpose of an agent is rather rudimentary: it passes along whatever requests it receives from its environment

(some client entity - not to be confused with the Client process of Chapter 2) to the bridge. In turn, it conveys whatever responses it receives from the bridge on to its environment. Thus, in the PS, the agents are simply one-element, bi-directional queues. Both the bridge and agent possess memories to store parameter values. The formal descriptions of these processes may be found in Appendix A.

3.1.1 Construction of Messages

Messages for both the bridge and the agents are organized around four distinct functions, illustrated in Figure 3.1: *create* or *c* to create a conference; *add* or *a* to add a target agent to a conference; *delete* or *d* to delete a target agent from a conference; and, *clear* or *l* to clear a conference. Each of the single letters representing functions are then suffixed with *q* for request, *i* for indication or success of request, and *r* for rejection or failure of request, forming the set of *message names* used in internal communication between bridge and agent. Those messages which are exploited for external communication between agent and environment are denoted instead by the double-letter suffixes, *qq*, *ii*, *rr*, for request, indication and rejection, respectively.

Two other ingredients complete the message. First, in order to distinguish messages by their origin or destination, agents (agent1 in Study 1; agent1 and agent2 in Study 2) are identified by appropriately installing a *1* or *2* at the head of the *message_names*. These identifiers also appear in agent *state_names*. In addition, it is recalled that a *?* signifies an input message event, and a *!* points to an output message event.

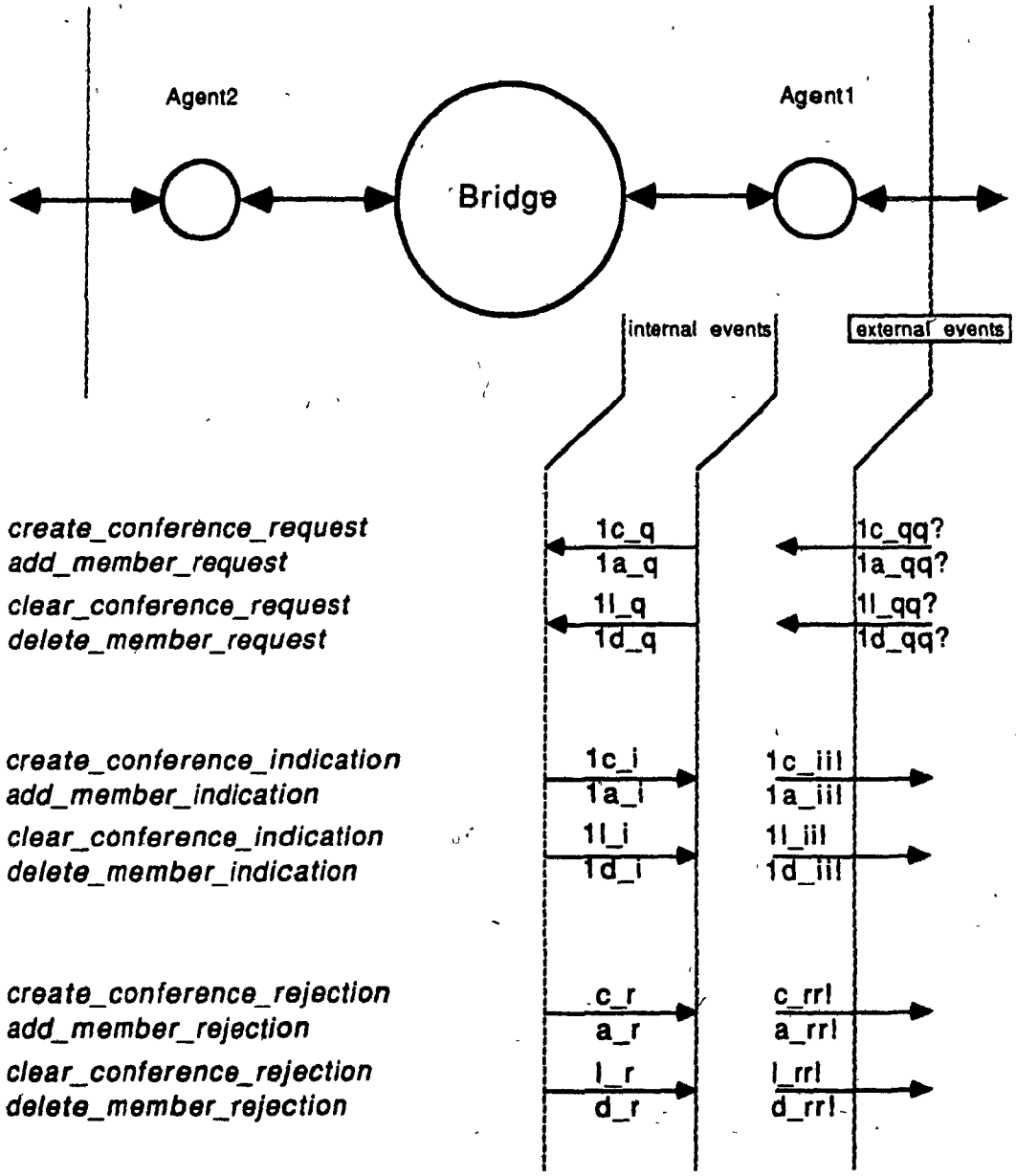


Fig. 3.1 Message Architecture for Preliminary System

3.1.2 Use of Parameters

Parameters are used to carry the identities of *target* agents, those agents requested to join or leave a conference initiated by agent1 or agent2. The identities of the target agents reside in the untyped set X and are specified as $x.1$ or $x.2$. In Study-2, agent1 can make agent2 a target for a conference initiated by agent1, and vice versa. The target agent parameter values are also used by the bridge, as explained later.

Reason code parameters, denoted as $c.1$, $a.1$, $d.1$, and $l.1$, being named after the four functions, respectively, accompany all rejection signals to explain the failure of a request. Reason codes are chosen from the counterpart sets C, A, D, *and* L, and may be alpha-numeric designations or actual narrative-form *reasons*. Examples of the latter, for the rejection of an *add* request may include:

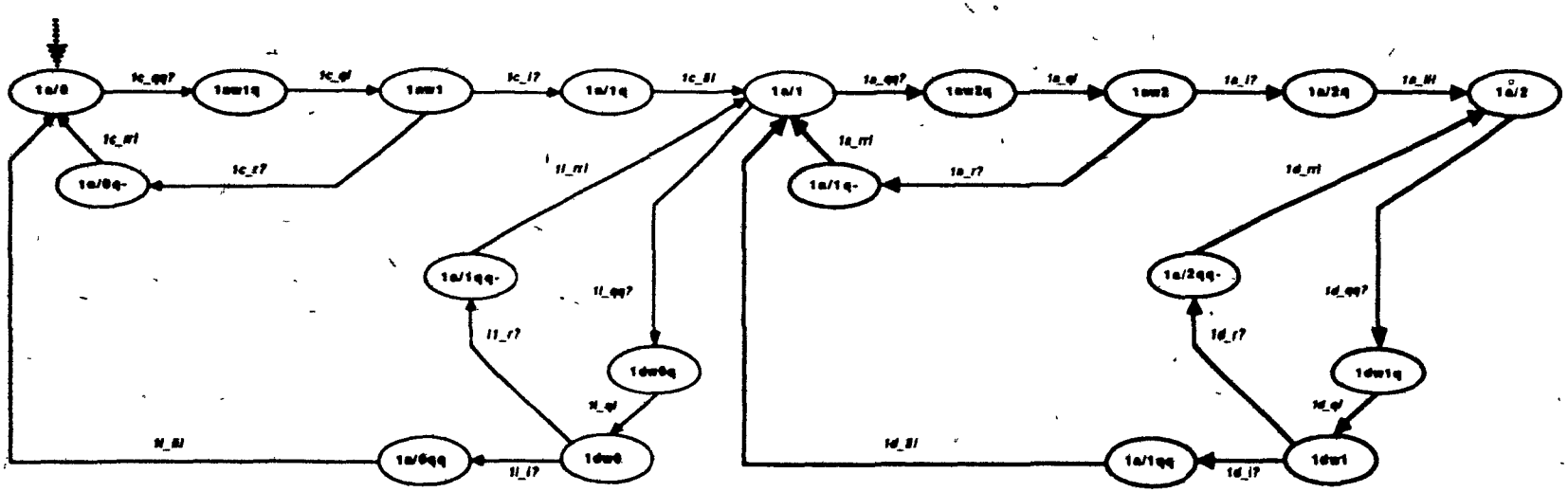
- " not added - target agent busy "
- " not added - target agent not responding "
- " not added - target agent already in conference "
- " not added - temporary bridge problem "

Such *reasons* would often be found in a real implementation. For this research, however, the reason code parameters accord a degree of completeness and flexibility to the formulations without specification of the actual contents of the reason code sets.

3.1.3 Description of Agent

The state-transition diagram for the agent process (agent1 or agent2) has 19 states and is illustrated in Figure 3.2. The process is entirely deterministic and has the three major states, `1a/0`, `1a/1`, and `1a/2`, which proceed as follows. From the empty state `1a/0`, a *create* request is expected from the environment. Once received, the agent enters a queue state, expressed with the letter `q` in state `1aw1q`. From `1aw1q`, the agent can transfer the internal version of the *create* request to the bridge. When the exchange is made, the agent enters the `1aw1` state, the first create/add wait mode. It progresses from the wait when it receives back from the bridge an indication or rejection to the request. Upon rejection, the process goes to another queue state, `1aw1q-`, from which it eventually delivers the rejection (with reason code) to its environment. The indication also leads to a queue state, `1a/1q`, where the indication is output to the environment, and the major state `1a/1` is reached: a conference has been created.

From the creation of a conference, it is possible to add a target agent to the conference or to dismantle the conference, depending on the reception of an *add* or a *clear* request from the environment. If the *add* request is received and successfully completed, the maximum capacity of two members is attained at `1a/2`. To continue, only a *delete* request is permissible. Note that the identity of the target agent specified in *add* and *delete* requests is kept in the requesting agent's memory, serving to verify that the response from the bridge to the request refers to the same target agent.



Notes: - parameters have been dropped from state and message names for ease of presentation
- see Appendix A for detailed process descriptions
- bold lines outline the add/delete module per agent of additional capacity for a conference
- agent2 is isomorphic

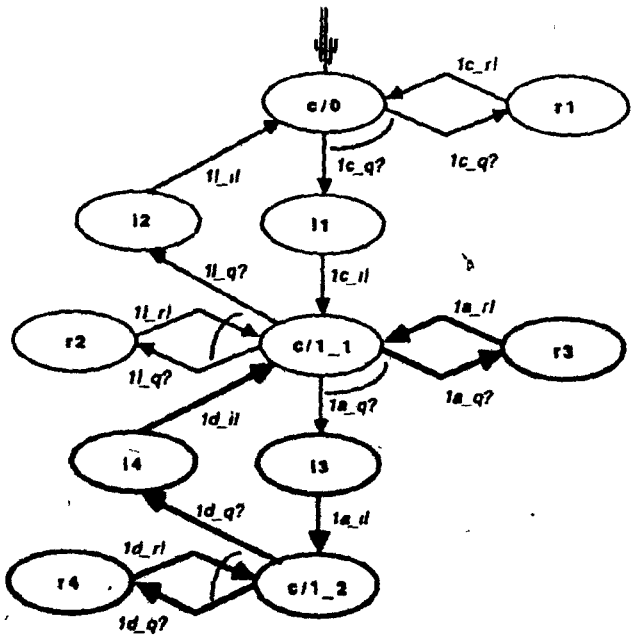
Fig. 3.2 State-Transition Diagram for Agent1 in Preliminary System

The *add*, *clear*, and *delete* procedures are completely analogous to the pattern followed for the *create* request. That is: reception of a request at the agent from the environment; exchange of the request between bridge and agent; bridge response of indication or rejection; progress to another major state on the former; and, loop to same major state on the latter. The *delete* and *add* procedures, taken together, form the module outlined in bold in Figure 3.2. Such modules can be replicated to obtain greater capacity for the agent process.

3.1.4 Description of Bridges

Two bridges are described in this subsection. The first has capacity for a single conference of two member agents and is able to communicate with a single agent only (agent1). The second, an extended version of the first bridge, has capacity for two conferences of two member agents each, and can communicate with two agents (agent1 and agent2). The state-transition diagrams of the single-conference and *extended* (two-conference) bridge processes are given in Figures 3.3 and 3.4, respectively.

The single-conference bridge has 11 states and reflects the behaviour of the agent, as would be anticipated for a relatively simple system. Once more, three major states are involved: `c/0`, `c/1_1`, `c/1_2`. `c/0` is the empty state, where no conferences are invoked. Since this bridge only communicates with agent1, `c/1_1` and `c/1_2` show the presence of a conference initiated by agent1, having one or two members, respectively. It is evident that this bridge process is modular, as outlined in bold in Figure 3.3.



Note - parameters have been dropped from state and message names for ease of presentation
- bold lines outline the add/delete module per agent of additional capacity for a conference
- arcs indicate paired deterministic selections
- see Appendix A for detailed process descriptions

Fig. 3.3 State-Transition Diagram for Single-Conference Bridge in Preliminary System

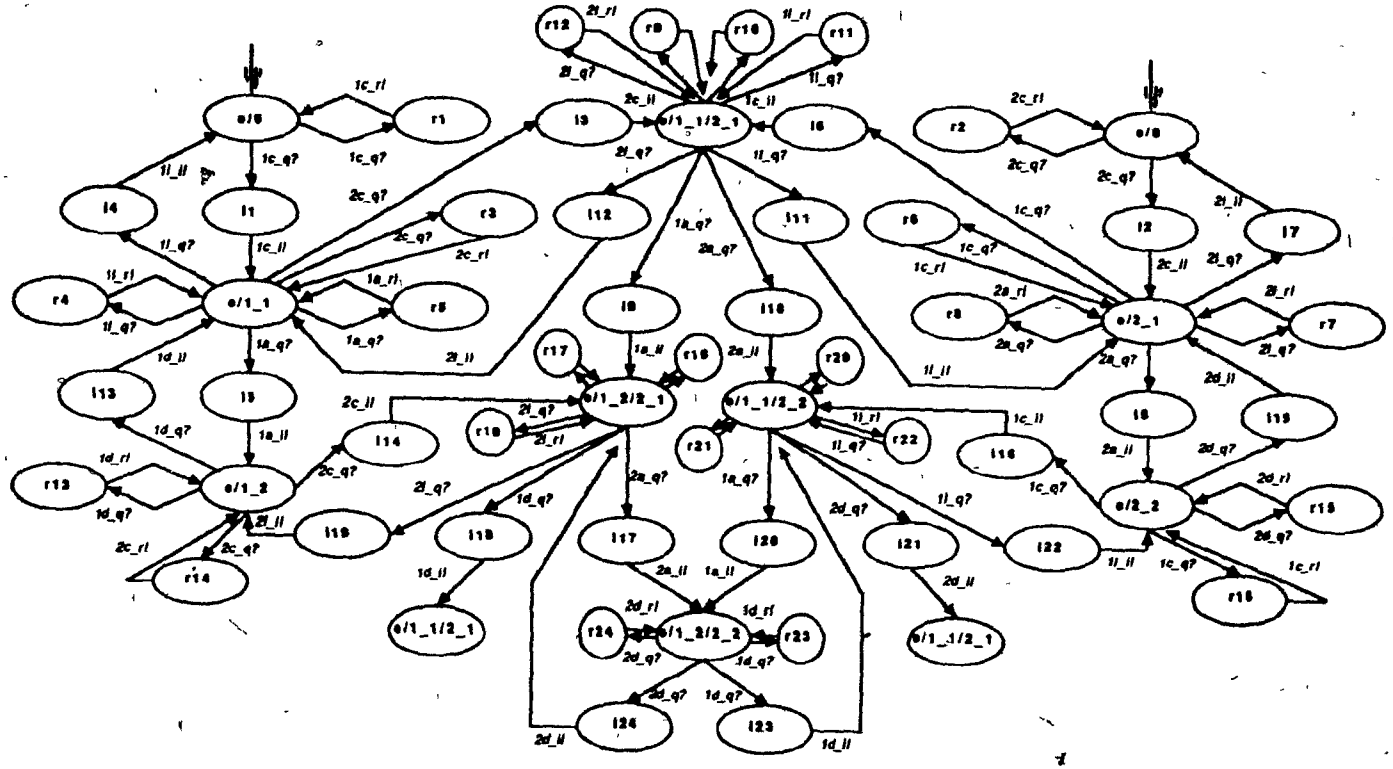


Fig. 3.4 State-Transition Diagram for Extended Bridge in Preliminary System

Other characteristics, including paired deterministic selections, are explained in the upcoming paragraphs.

The structure of the *extended* bridge is more complicated and a simple modular construction is difficult to see. However, an attractive symmetry may be observed, as though two single-conference systems had been configured together. The major states of the simple bridge are augmented by c/2_1, c/2_2, c/1_1/2_1, c/1_1/2_2, c/1_2/2_1, and c/1_2/2_2 in the *extended* bridge. Thus, the *extended* bridge now indicates in its major *state_names* the existence of one or two conferences, the identities of the corresponding initiators of those conferences, and the number of member agents in each conference. Following is state (39) of 57 states, extracted from the *extended* bridge process description, to exemplify this notation and give some flavour of the operation of the bridge:

```
(39) c/1_1/2_2(x.1) =
        + #(x.2) . 1a_q?(x.2) ; i20(x.1,x.2)  --> (46)
        + #(x.2) . 1a_q?(x.2) ; r20(x.1,x.2)  --> (47)
        + 2d_q?(x.1) ; i21(x.1)  --> (48)
        + 2d_q?(x.1) ; r21(x.1)  --> (49)
        + 1l_q?  ; i22(x.1)  --> (50)
        + 1l_q?  ; r22(x.1)  --> (51)
```

Here, the *state_name* c/1_1/2_2 indicates that agent1 has created a conference (labelled by the first 1) with itself as the single member (quantified by the second 1), and agent2 has created a conference (labelled by the first 2) with itself and a

target agent as members (quantified by the second 2). The identity of the additional member of agent2's conference is given by the parameter $x.1$.

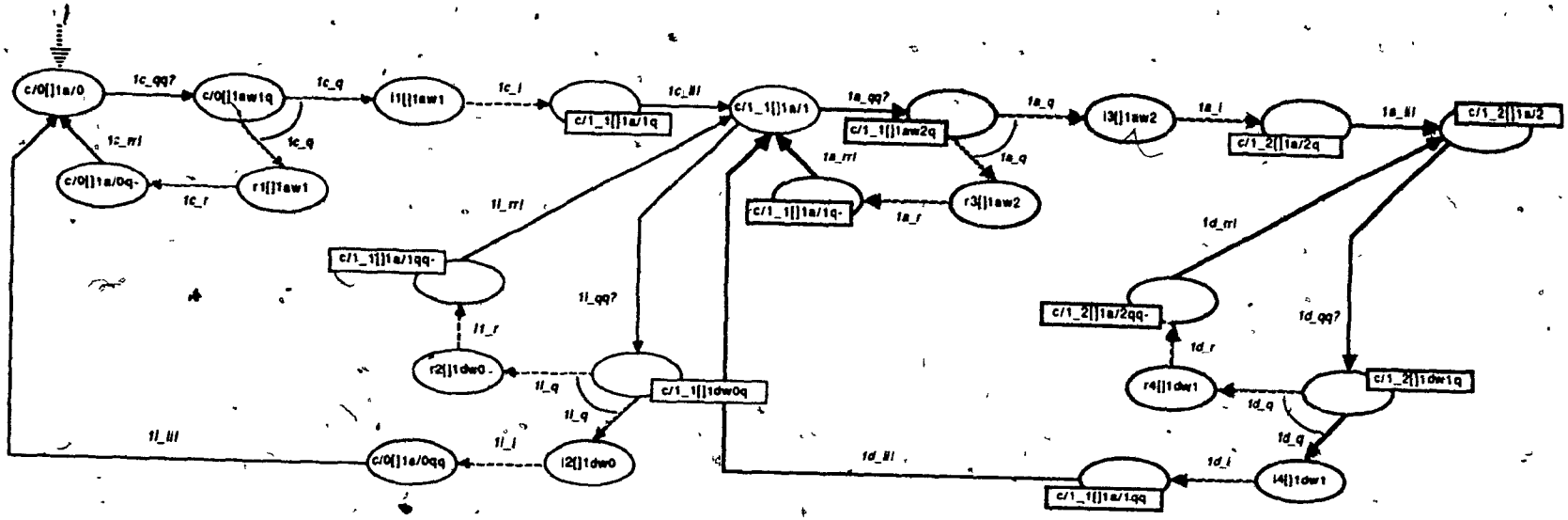
The above transition equation allows the bridge to accept one of three possible request messages, each of which has a pair of *deterministically selected* successor processes, modelled in a non-deterministic fashion and marked by arcs in the figures. These special pairings, which follow the absorption of a request, are due to binary decisions taken by the bridge on the basis of parameter values or other information. The first pair of successors is attributable to an *add* request from agent1, with the parameter $x.2$ as the identity of the target agent to be added to agent1's existing conference. The successor processes, `i20` and `r20`, are branchings generated from a bridge decision to accept or reject the request. Parameter $x.2$, along with the original parameter $x.1$, must be retained in memory for further manipulation, such as subsequent *delete* requests. The next pair of branchings arise from a *delete* request from agent2. The identity of the (only) member to be deleted from agent2's conference is found in the message as the parameter $x.1$ (note that agent1 and agent2 can only add or delete from their own respective conferences). This value is referenced to the *global_binding* appended to the *state_name*. Again, branchings occur on success or failure of the request, with the $x.1$ parameter being dropped once an indication is passed on to agent2. Similarly, the last pair results from a *clear* request from agent1 to drop the conference it had previously created. This request does not have any parameters, but the successor processes must guard $x.1$ since $x.1$ is pertinent to agent2's conference, which is unaffected by the request.

The three failures loop back to state (39), while (46) eventually goes to the major state c/1_2/2_2(x.1,x.2) - or state (52) - where the parameters have been renamed to preserve their association with the conference ordering in the *state_name*. The bridge reaches maximum capacity at state (52), from which only deletions are acceptable. States (47) and (50) lead to the major states c/1_1/2_1 and c/2_2(x_1), respectively.

3.2 Study 1: Single-Conference System

In this study, the bridge process, the agent process named agent1, and their interactions are explored. As mentioned, this bridge has capacity for one conference of two member agents, and agent1 has capacity for one member agent in addition to itself.

First, the []-product, as defined in Chapter 2, is applied to the two processes. The resultant contains 23 states and is portrayed in Figure 3.5. Alternatively, the algebraic output is available in Appendix A. Within the compound states, the left side of the [] gives the state of the bridge process, while the right gives the state of the agent1 process. The two processes progress together, without deadlocks. That is, neither process suffers an indefinite wait due to the absence of a desired message exchange. Parameters are found to be correctly carried and passed within and between processes, when compared to hand-worked examples. The memory available to both the bridge and agent1 allows for faithful internal exchanges, eliminating the possibility of errors in the communication channel.



Notes
- parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- broken lines and single-letter suffixes in messages indicate internal events
- bold lines outline the add/delete module per agent of additional capacity for a conference
- see Appendix A for detailed process descriptions

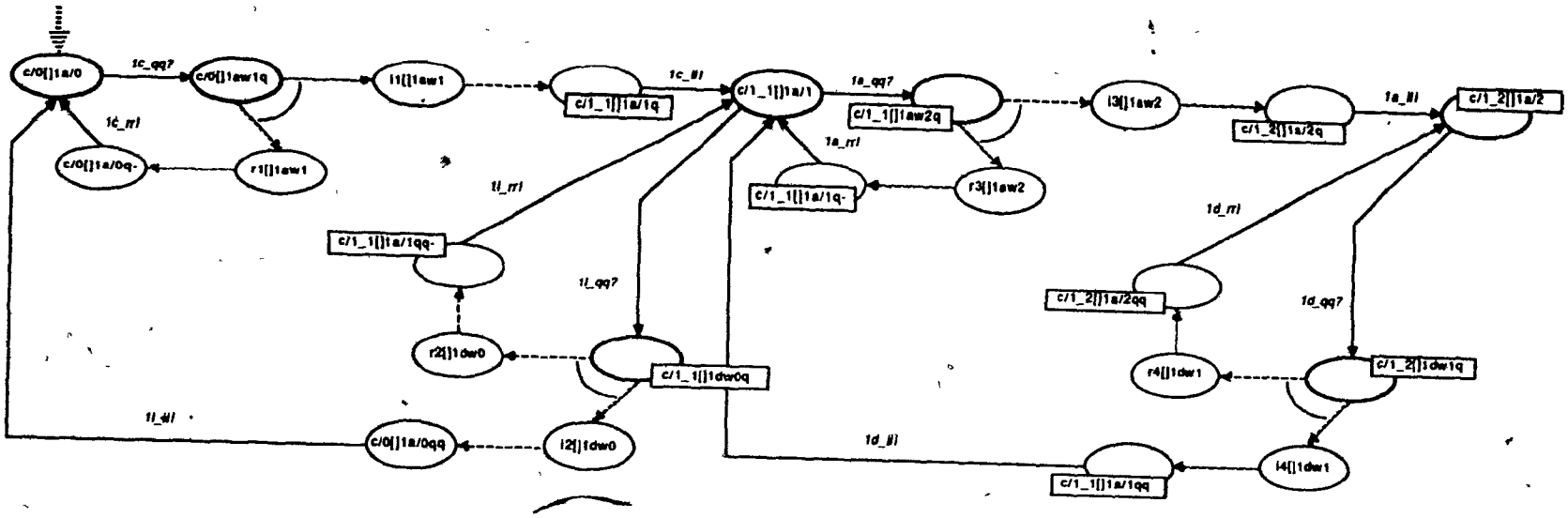
Fig. 3.5 State-Transition Diagram for []-Product for Single-Conference Preliminary System

The modular structure of the ||-product is highly evident and is, of course, a direct consequence of the modularity of the bridge and agent1 processes. Therefore, the capacity of the conference system may easily be increased by appending more *add/delete* modules (outlined in bold in Figure 3.5), a very desirable feature. For a conference capacity of N agents, then, a total of N-1 modules would be necessary.

The paired *deterministic selections* of the ||-product are also inherited from the bridge process. As with the bridge, there are four pairings in all: one each for the *create*, *clear*, *add*, and *delete* requests.

Although there is freedom from deadlocks, certain looping behaviours (i.e. live-locks) are imaginable. For example, repeated arrival from the environment of an *add* request with the same inadmissible target agent (identified in the parameter), would result in the rejection loop: `c/1_1[]1a/1` ... `c/1_1[]1aw2q` ... `r3[]1aw2` ... `c/1_1[]1a/1q-` ... `c/1_1[]1a/1`. Normally, it would be preferable to limit the number of retry-rejection cycles to some small number, say three. As such constraints are not imposed here, it may be said that this is a *patient* system.

To proceed with the interconnection of the two processes, the R-operator is now engaged to *hide* the internal or trace events. The effect is rendered in Figure 3.6 and is provided in algebraic form in Appendix A. Internal events have been removed and states which are externally visible have been made bold. Externally invisible states and transitions have been left intact for reference.



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- see Appendix A for detailed process descriptions

Fig. 3.6 State-Transition Diagram for R for Single-Conference Preliminary System

At this stage, it would be advantageous to remove R-equivalent states to reduce the R of the conferencing system further. A set of equivalence assumptions were made by the tools and inductively tested over two passes. Four equivalences were discovered, these being:

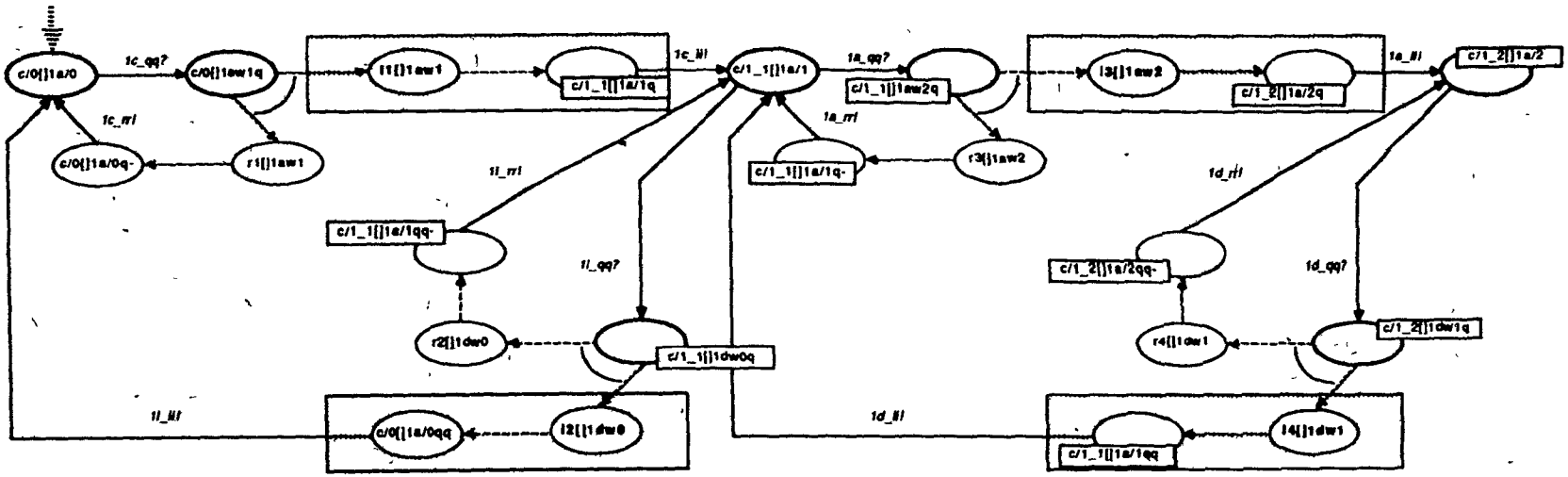
$$R(c/1\_1[]1a/1q) \equiv R(i1[]1aw1)$$
$$R(c/1\_2(x.1)[]1a/2q(x.1)) \equiv R(i3(x.1)[]1aw2(x.1))$$
$$R(c/0[]1a/0qq) \equiv R(i2[]1dw0)$$
$$R(c/1\_1[]1a/1qq(x.1)) \equiv R(i4[]1dw1(x.1))$$

This simplifies the original system to 19 states (see Figure 3.7). Since the remaining internal events do not meet the stability criterion of Chapter 2, the system is said to be unstable.

Finally, if the *reason code* parameter sets involved in rejection procedures were collapsed to have cardinalities of one, the R-equivalence would also hold for the four pairs of states:

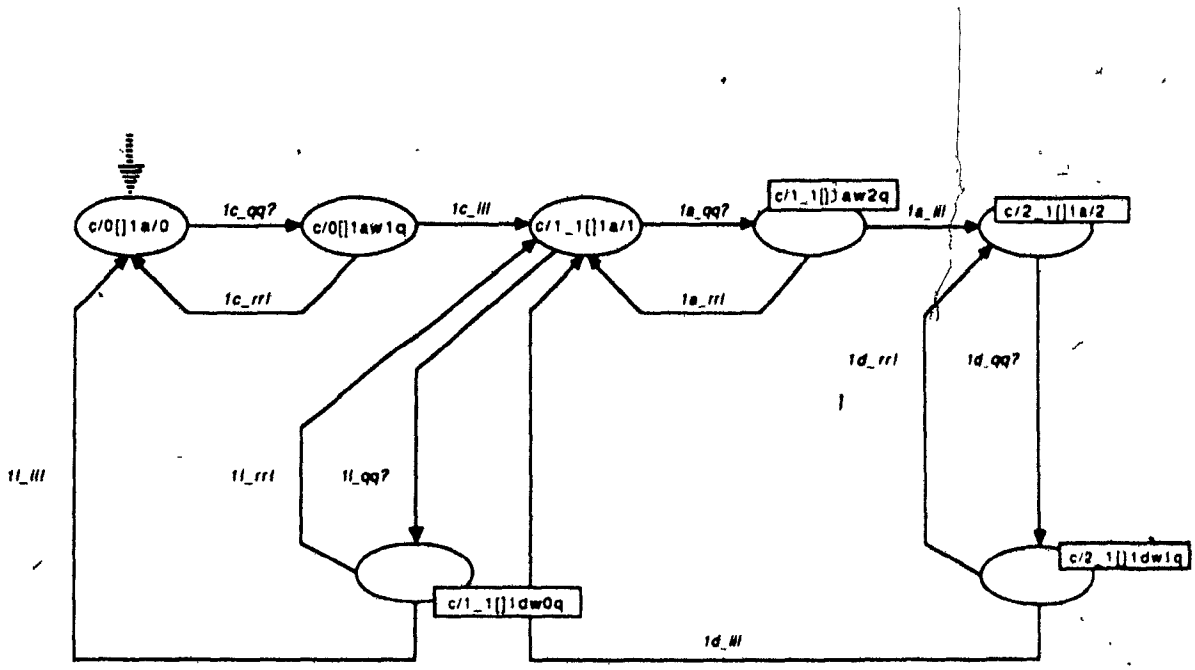
$$R(c/0\ []1a/0q-\ ) \equiv R(r1[]1aw1)$$
$$R(c/1\_1[]1a/1q-\ ) \equiv R(r3[]1aw2)$$
$$R(c/1\_1[]1a/1qq-) \equiv R(r2[]1dw0)$$
$$R(c/1\_2[]1a/2qq-) \equiv R(r4[]1dw1)$$

This would remove another source of instability. Figure 3.8 gives the *constrained* externally visible behaviour of this conferencing system. It remains an unstable sys-



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- boxed pairs of states are equivalent (the state with name in bold replaces other one)
- see Appendix A for detailed process descriptions

Fig. 3.7 State-Transition Diagram for R-reduced for Single-Conference Preliminary System



Notes: - parameters have been dropped from state and message names for ease of presentation
- see Appendix A for detailed process descriptions

Fig. 3.8 State-Transition Diagram for Constrained Single-Conference Preliminary System

tem due to the four deterministic pairings for indication and rejection.

The combined processing for the []-product, R. equivalence tests, and reductions, using the automated DCP tools, took 78 seconds of CPU time. Page faults, which are a measure of the memory requirements for the LISP environment, amounted to 20.7k (k = 1000).

3.3 Study 2: Two-Conference System

The two-conference PS formulation consists of a bridge extended to a capacity of two conferences of two member agents each, and two distinct but isomorphic agents (agent1 and agent2) each with capacity for one member agent in addition to themselves. This *extended* bridge was depicted in Figure 3.4; agent1 and agent2 are as previously given in Figure 3.2. Computer processing capability restricts this analysis to the []-product.

The []-product of the *extended* bridge communicating with agent1 and agent2 yields 465 states in total, without deadlocks. Processing time was 20:39 minutes; page faults exceeded 278.7k.

The capacity of the bridge and agent processes can certainly be increased to attain quite large conferencing systems. Although it is possible to do so, the formulation would quickly become unwieldy.

Chapter 4 The Membership-Based System

The Membership-Based System (MBS) is introduced to overcome various inherent limitations of the PS formulation. Some of these limitations, such as the cumbersome need for instantiated *message_names* and *state_names* to distinguish agent processes, are already evident. Other drawbacks become obvious when an attempt is made to widen the PS formulation to provide larger-capacity and more flexible conference systems. Quickly, the tracking of conferences and their membership lists, as well as the provision of more advanced request facilities becomes quite imposing. From these impediments surfaces a desire for compact yet highly expressive notation, which is met chiefly through expanded use of parameterization. Concurrently, several simplifications are made to the message architecture. The fruit of these efforts is a set of modular processes which are easily generalized to form conferencing systems of arbitrarily large capacity. Again, two studies are recounted: a single-conference system of a bridge and one agent, and a two-conference system of the same bridge with two agents.

4.1 Formulation of Processes

The bridge and agent processes follow the same guiding principles as in the PS formulation. The bridge has capacity for up to four agents in any combination of conference groups. The agent supports a maximum of two agents (including itself) in a single conference and has some new duties. All requests are designed such that only one agent is added to or deleted from the total membership of the conference system at any given time, hence the name Membership-Based System.

Both the bridge and agent processes are deterministic. The only true non-determinisms are in the arrival of request messages from the environment, and in the parameter values originating with those events. Occasionally, these determinisms and non-determinisms are nested together and require careful scrutiny in the []-product and other results.

4.1.1 Construction of Messages

The PS formulation defined messages around four functions: *create*, *add*, *delete* and *clear*. As a simplification, the present formulation reduces this number to two. This reduction is realizable given the greater use of parameterization. Thus, the *message_names* corresponding to the *create* and *add* functions (i.e *c_q*, *c_i*, *c_r* and *a_q*, *a_i*, *a_r*, respectively) in both the bridge and agent process descriptions are merged into the *add* messages. Similarly, the messages for *delete* and *clear* are amalgamated into the group of *message_names* based on the *delete* function. Of course, the changes

also affect the external messages. This new message format is illustrated in Figure 4.1. Functional distinction is now provided through the bridge's interpretation of the interrelationships of three parameter values supplied in all requests: the *requestor*, the *target*, and the *conference name*. These parameters are described further in the next subsection; their interpretation by the bridge is left for discussion in Section 4.1.4.

The above simplification may be carried further to leave just one function or even one *message_name*, embedding all information in parameters. Although compactness is gained, there is a trade-off to be juggled, as increased parameterization makes for difficult tracking of processes. Concomitantly, the processes must become more complex to decipher the information in the parameters.

Message_names no longer bear a *1* or *2* to inform of a message's origin or destination with respect to agent identities, this service now being provided by the *requestor* parameter. The *?* and *!* symbols have the same significance as before.

4.1.2 Use of Parameters

The MBS formulation calls on parameters to perform several new services. Both the MBS-bridge and the MBS-agent utilize the *target* parameter, renamed $y.1$ or $y.2$, discussed previously. For the bridge, the *requestor* or $x.1$, *conference name* or $n.1$, and the *grouping table* or $gn.1$ parameters are introduced; the agent is given corresponding *requestor*, *conference name* and *membership list* or $mn.1$ parameters.

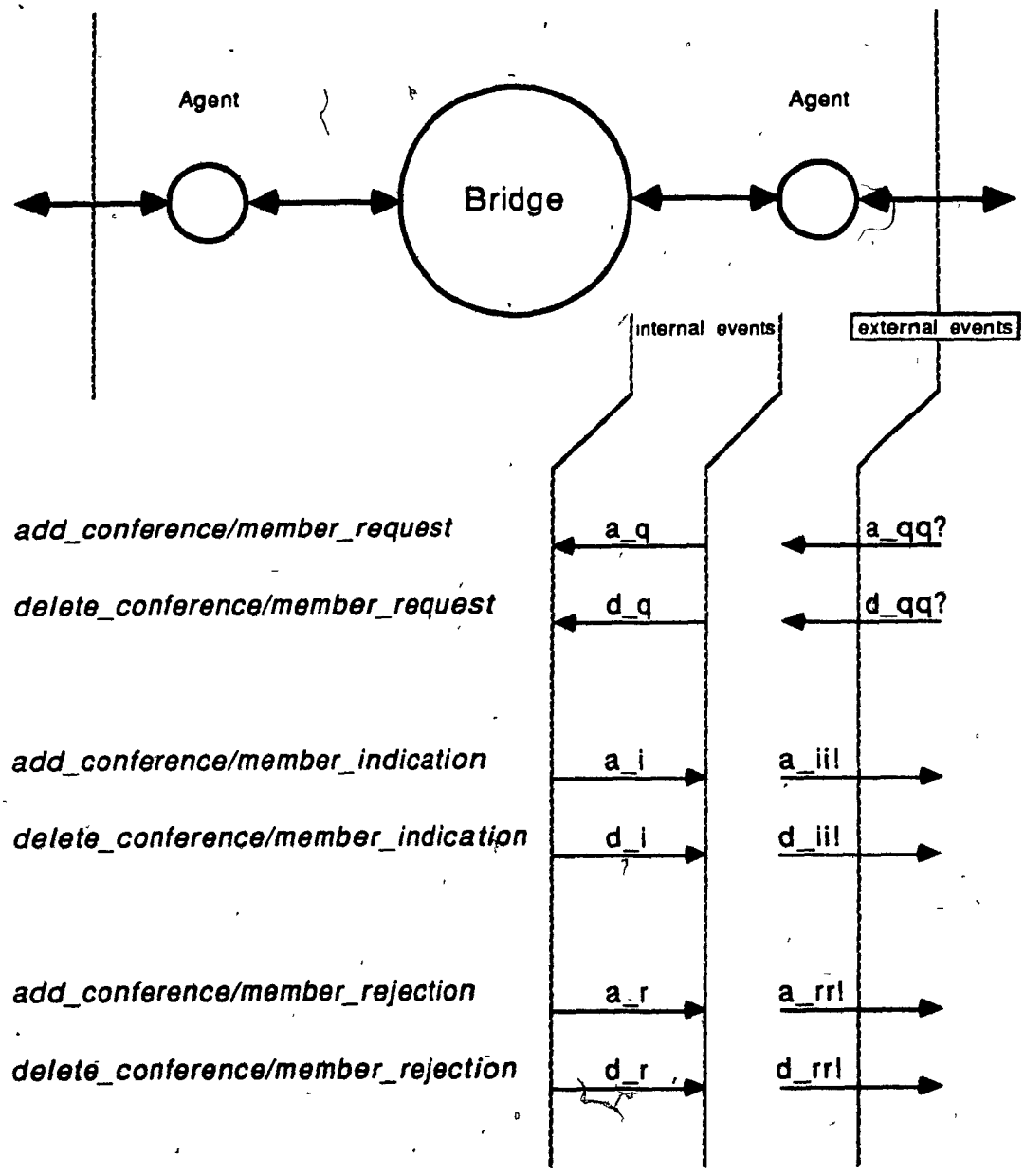


Fig. 4.1 Message Architecture for Membership-Based System

The *grouping table* and *membership list* parameters prevent the linear growth of the number of member agent parameters with the increasing capacity of bridge and agent processes, respectively. All parameters are based on untyped sets.

The *requestor* parameter, which contains the identity of the agent submitting a request to the bridge, is fixed for each agent at execution time by a de-conflicting algorithm within the automated tools. The *requestor* identity is written onto messages received from the environment, and stripped from messages delivered to the environment by the appropriate agent. The *conference name* parameter is used to symbolically refer to a particular conference, leading to richer request facilities for the agent while necessitating more advanced logic within the bridge to handle such requests.

The *grouping table* parameter plays a critical role for the bridge, tracking conferences by maintaining multiple membership lists with respective conference names and initiator (chairman) identities. The result is a very compact and modular representation of an otherwise messy affair. With this super-parameter, it is possible to generalize the bridge process description to handle any number of agents in any grouping patterns. For example, if the parameter stands at *g*2.1, it means that there are currently two agents being tracked by the bridge, either as two initiator-dependent situations of a single conference of two agents, or as two conferences of one agent each. Externally, the distinction is not apparent, except by observation of the states of initiating agents. When another agent is to be *added*, the parameter proceeds

to $g3.1$. Based on the request and the preceding grouping configuration, there may now be three conferences of one agent each, two initiator-dependent situations of two conferences with two agents in one and one in the other, or two initiator-dependent situations of three agents in a single conference. This implies that a 1:1 mapping exists such that $gj+1.1$ is created given the parameters $gj.1$, $x.1$, $y.1$, and $n.1$. The inverse of this mapping must also be available for *deletes*. This mapping is simulated in the process descriptions simply through the displacement of the preceding *grouping table* parameter by the current one at appropriate points.

Lastly, the *membership list* parameter invoked in the agent, although not absolutely necessary to the operation of the agent, provides for local retainment of the identities of agents currently engaged in the conference (if any) which that agent has initiated. The list is enlarged or diminished in an orderly way, from $mj.1$ to $mj+1.1$ or mj-1.1, one agent at a time, respectively. The *membership list* does give a means of cross-checking the agent states with the bridge states in the []-product, and is an aid in the generalization of the MBS-agent process description. It is conceivable that a more *intelligent* agent could make use of the membership list to perform preliminary validity checks on requests before sending them on to the bridge.

Reason code parameters are employed as in the PS formulation. Following the simplifications made to the *message_names*, they are now based over the sets *A* and *D*.

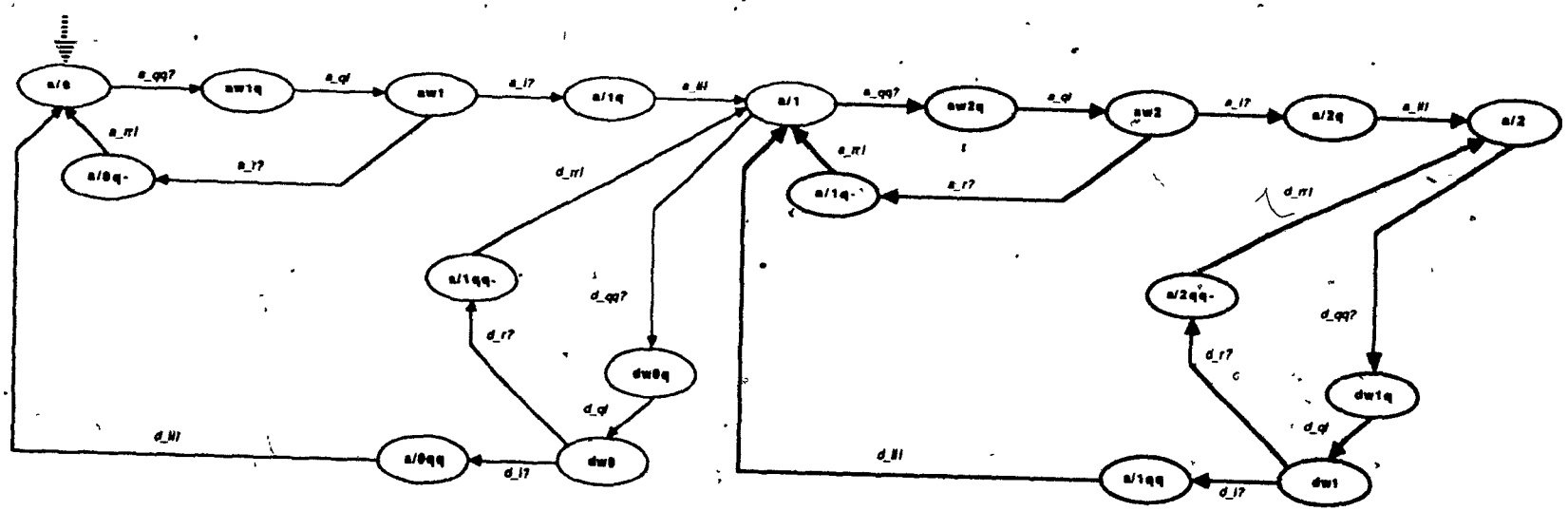
4.1.3 Description of Agent

The state-transition diagram for the agent process, with capacity for one member agent in addition to itself, has 19 states and is given in Figure 4.2. Algebraic descriptions are provided in Appendix B. The MBS-agent mirrors the modular structure and follows the same basic pattern of behaviour of the PS-agent, which also had 19 states. Differences may be observed in the merger of *message_names*, the absence of agent identifiers appearing in *state_names* and *message_names*, and the greater use of parameters, as discussed earlier. The identity of the target agent, as well as the conference name, specified in *add* and *delete* requests are kept in memory, to correlate the response from the bridge with the contents of the original request. The identity of an agent, in the form of the requestor parameter, is passed from state to state such that the agent always retains its name.

With the above modifications, a single, generic agent is made available which can be replicated as many times as desired without ambiguity. When two agent processes are used in the []-product for the Two-Conference System, the processes are de-conflicted by the automated tools. This archetype agent process description is generalizable to any membership capacity, with the number of states totalling 9A+1 to handle A agents.

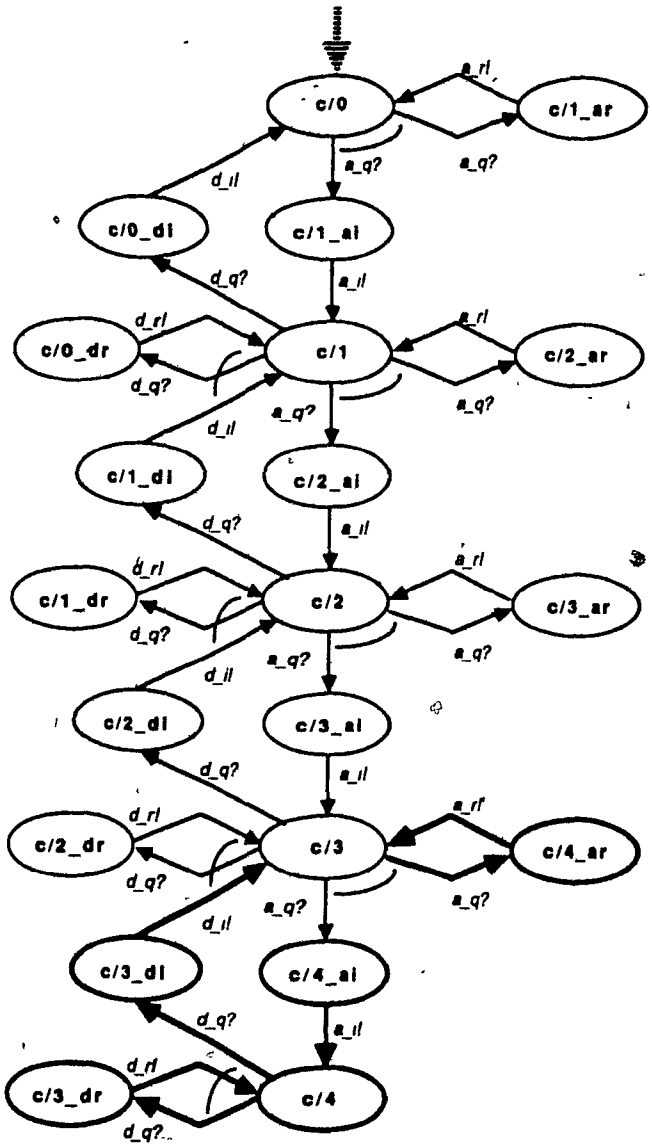
4.1.4 Description of Bridge

The state-transition diagram of the bridge process has 21 states and is pictured in Figure 4.3, with algebraic expressions to be found in Appendix B. The MBS-bridge



Note: - parameters have been dropped from state and message names for ease of presentation
- bold lines outline the add/delete module per agent of additional capacity for a conference
- see Appendix B for detailed process descriptions

Fig. 4.2 State-Transition Diagram for Agent in Membership-Based System



Notes:- parameters have been dropped from state and message names for ease of presentation
- bold lines outline add/delete module per agent of additional capacity for a conference
- see Appendix B for detailed process descriptions

Fig. 4.3 State-Transition Diagram for Bridge in Membership-Based System

can support any conference groupings for up to four agents, replacing the two PS-bridges of 11 and 57 states by one process. Thus, a spectrum of possibilities exists from a single conference of four agents at one end, to four conferences of one agent each at the other extreme.

The MBS-bridge has the five major states, c/0, c/1, c/2, c/3, c/4, and up to c/2, has the same form as the basic PS-bridge. Once more, c/0 is the empty state, where no conferences have as yet been constructed. The remaining major states indicate the total number of agents participating in all conferences, where the actual configuration is furnished in the *grouping table* parameter, *g*n.1. It is evident that this bridge process is again modular, as outlined in bold in Figure 4.3, and generalizable with 5A+1 states for a capacity of A agents.

As an example and to draw comparisons with the *extended* PS-bridge, state (13) of the MBS-bridge process description is selected:

```
(13) c/3(g3.1) =
        + #(x.1,y.1,n.1) .  a_q?(x.1,y.1,n.1) ; c/4_ai(x.1,y.1,n.1,g3.1)  --> (14) .
        + #(x.1,y.1,n.1) .  a_q?(x.1,y.1,n.1) ; c/4_ar(x.1,y.1,n.1,g3.1)  --> (15)
        + #(x.1,y.1,n.1) .  d_q?(x.1,y.1,n.1) ; c/2_di(x.1,y.1,n.1,g3.1)  --> (16)
        + #(x.1,y.1,n.1) .  d_q?(x.1,y.1,n.1) ; c/2_dr(x.1,y.1,n.1,g3.1)  --> (17)
```

Here, the *state_name* c/3 indicates that some conference(s) exist(s) with a total of three agents enrolled altogether. Instead of the six transitions of state (39), only four are available. All four are of similar form, with the three parameters (x.1,y.1,n.1)

drawn from *local_bindings*. The above transition equation allows the bridge to accept one of two possible request messages, each of which has a pair of *deterministically selected* successor processes (as described previously in Chapter 3). The first pair of successors is attributable to an *add* request from a *requestor* $x.1$, with the target agent to be added $y.1$ under *conference name* $n.1$. The successor processes, `c/4_ai` and `c/4_ar`, are branchings generated from a bridge decision to accept or reject the request, as the suffixes suggest. All of the incoming parameter values, along with the original parameter $g3.1$, must be retained for the response. Once the response is emitted, only the $gn.1$ parameter is held for subsequent actions such as *delete* requests. The second pair of branchings arise from a *delete* request, carrying three parameter values as above. Again, branchings occur on success or failure of the request.

The two failures loop back to state (13), while (14) eventually goes to the major state `c/4`($g4.1$), or state (18). The bridge reaches maximum capacity at state (18), from which only deletions are acceptable. (16) travels to the major state `c/2`($g2.1$), or state (8).

As has been alluded, the bridge has a considerable decision-making task upon receipt of a request. In order to formulate a response, whether an indication or a rejection, with the appropriate *reason code* justifying and explaining a rejection, a number of atomic tests must be performed. An example flowchart is provided in Figure 4.4, which demonstrates the internal logic the bridge must carry out in order to decide among four successful and six rejection cases for an *add* request. Two of the

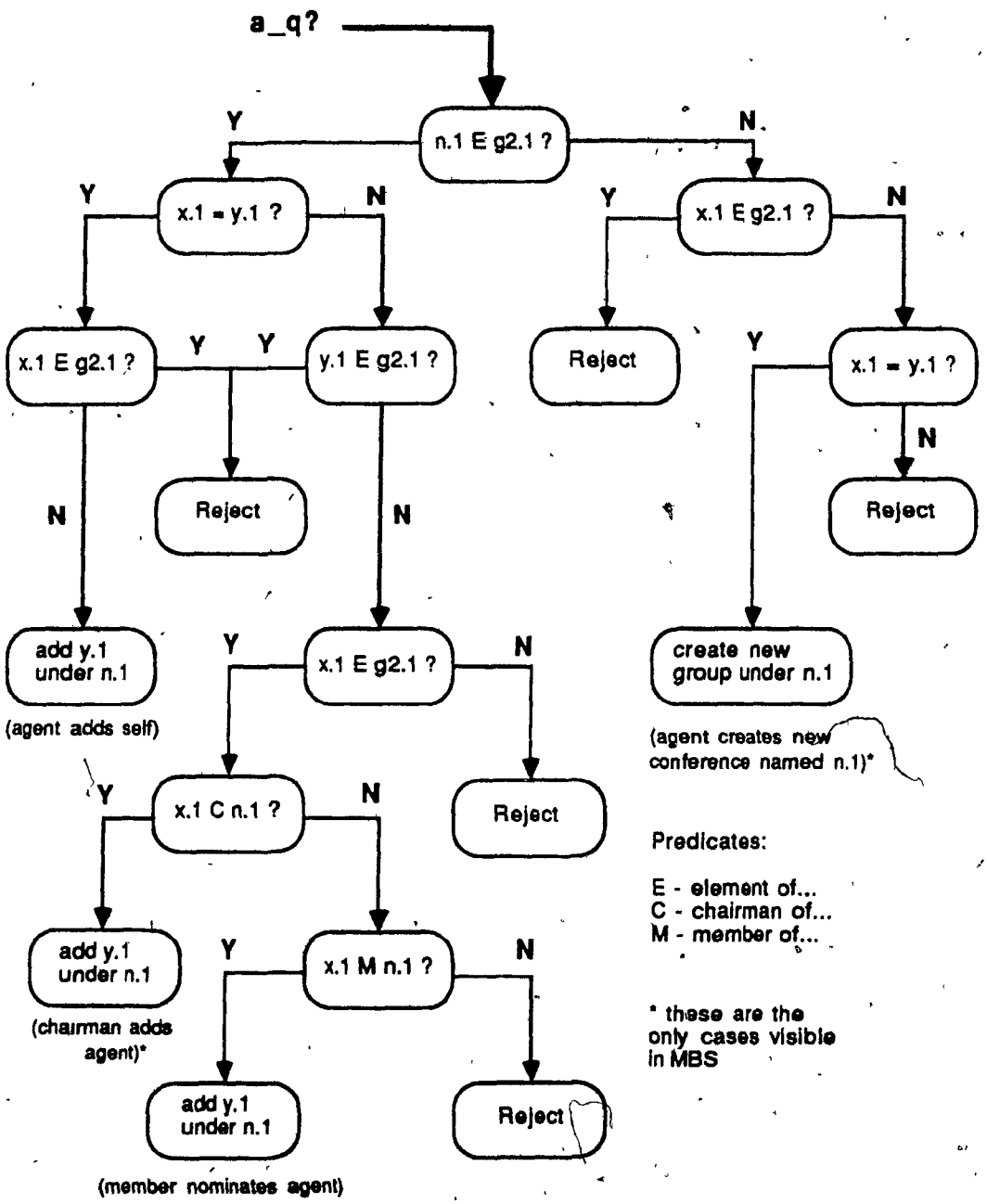


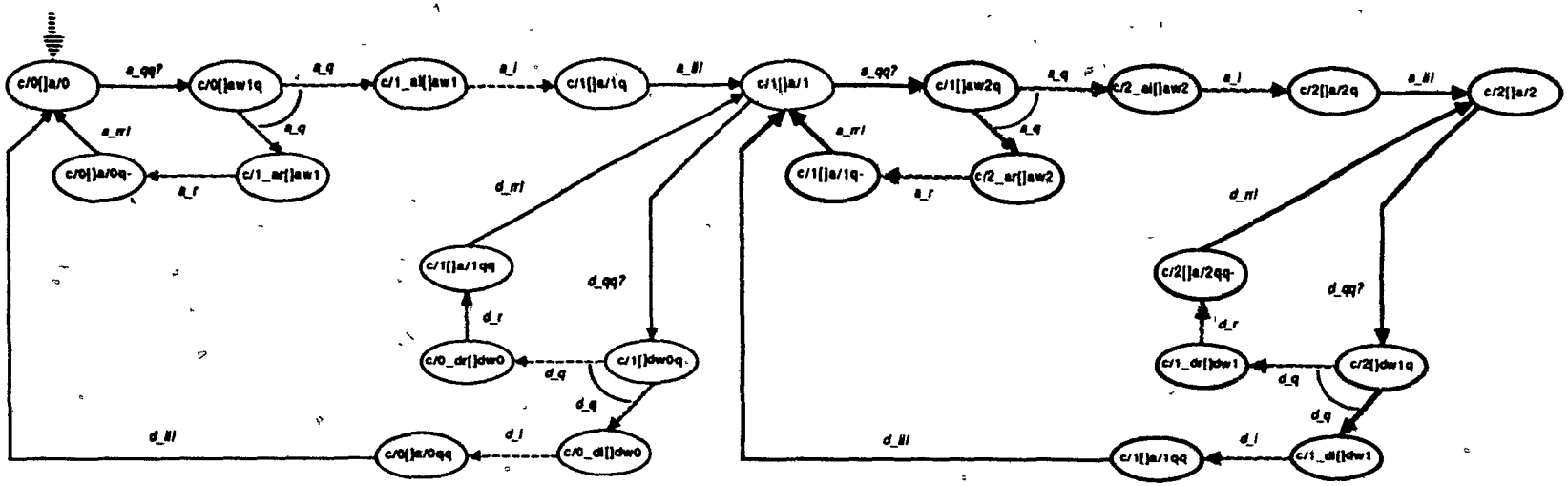
Fig. 4.4 Internal Logic for MBS-Bridge - Example of *Add* Feature with Disjoint Sets

successful *add* cases emerge in the current formulation: *agent creates new conference named* $n.1$; and, *chairman adds agent* (a DIAL-OUT connection scenario). The other two cases require further facilities to implement. For the first of these cases, where a conference is initially created, it is necessary to meet the conditions: $x.1 = y.1$ (i.e. *requestor* and *target* are one and the same); and $x.1$, $n.1$ not contained in the *grouping table* $g2.1$ (i.e. the *chairman*-to-be, $x.1$, and the *conference*-to-be, $n.1$, are new and uniquely-named). In the second case, where a conference is to be augmented with a new member, $x.1 \neq y.1$, $y.1$ is unknown to the *grouping table*, $x.1$ is known to the *grouping table*, and $x.1$ is the *chairman* of *conference* $n.1$. Many other interpretations, accompanied by the appropriate logical predicates, may be envisioned.

4.2 Study 1: Single-Conference System

This study examines the bridge process in communication with a single agent process. Many similarities are found between this Single-Conference System and the corresponding one of Chapter 3.

The computation of the []-product arrives at 23 states (as was the case with the []-product of Study 1 in Chapter 3) and is discovered to be free of deadlocks. The result, in graphic form, is given in Figure 4.5, while the algebraic output is available in Appendix B. Again, parameters are found to be correctly carried and passed within and between processes. The temporary storage by the bridge and requesting agent processes of all parameter values appearing in a request assures error-free internal



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- broken lines and single-letter suffixes in messages indicate internal events
- bold lines outline the add/delete module per agent of additional capacity for a conference
- see Appendix B for detailed process descriptions

Fig. 4.5 State-Transition Diagram for []-Product for Single-Conference Membership-Based System

exchanges for responses, since a match must be achieved between the two sets of parameters.

The modular structure of the []-product, as well as the inheritance of paired *deterministic selections* and the possibility of livelocks, are as observed in the Single-Conference System of Chapter 3.

The application of the R-operator to the []-product follows. The effect is rendered in Figure 4.6 and is provided in algebraic form in Appendix B. It would be helpful to remove R-equivalent states to reduce the R of the conferencing system further. A set of equivalence assumptions were made by the tools and inductively tested over two passes. The four equivalences detected are listed below:

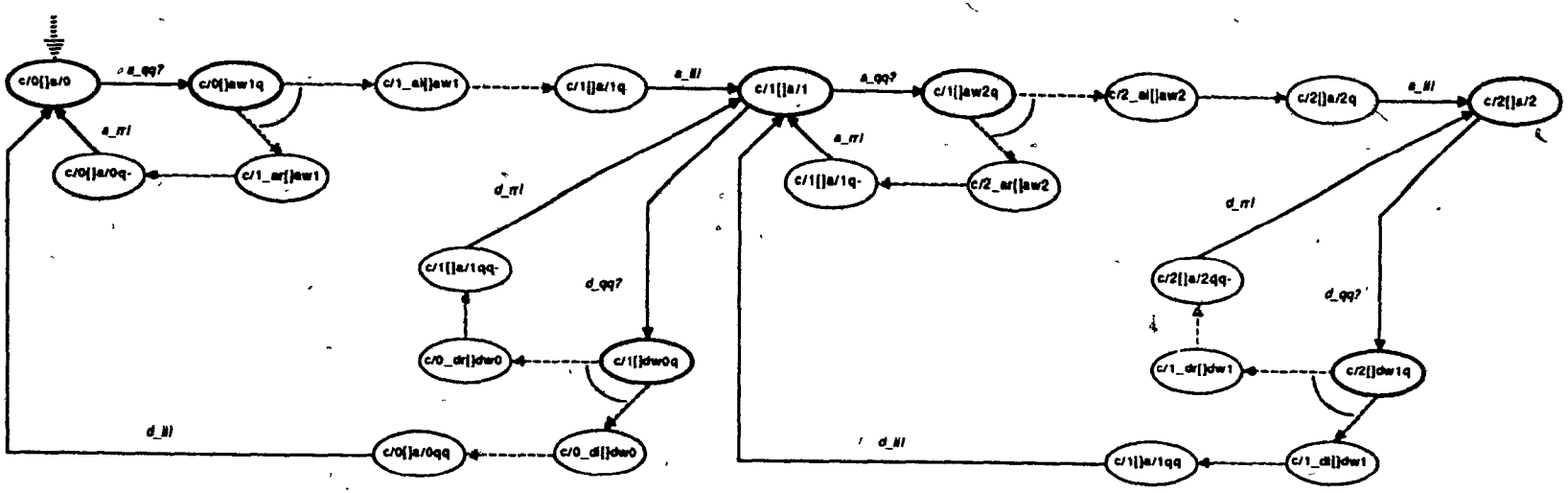
$$\mathtt{R(c/1(}g1.1\mathtt{)[]a/1q(}x.1,y.1,n.1\mathtt{))} \equiv \mathtt{R(c/1\_a1(}x.1,y.1,n.1\mathtt{)[]aw1(}x.1,y.1,n.1\mathtt{))}$$

$$\mathtt{R(c/2(}g2.1\mathtt{)[]a/2q(}x.1,y.1,n.1,m1.1\mathtt{))} \equiv \mathtt{R(c/2\_ai(}x.1,y.1,n.1,g1.1\mathtt{)[]aw2(}x.1,y.1,n.1,m1.1\mathtt{))}$$

$$\mathtt{R(c/0[]a/0qq(}x.1,y.1,n.1\mathtt{))} \equiv \mathtt{R(c/0\_di(}x.1,y.1,n.1,g1.1\mathtt{)[]dw0(}x.1,y.1,n.1,m1.1\mathtt{))}$$

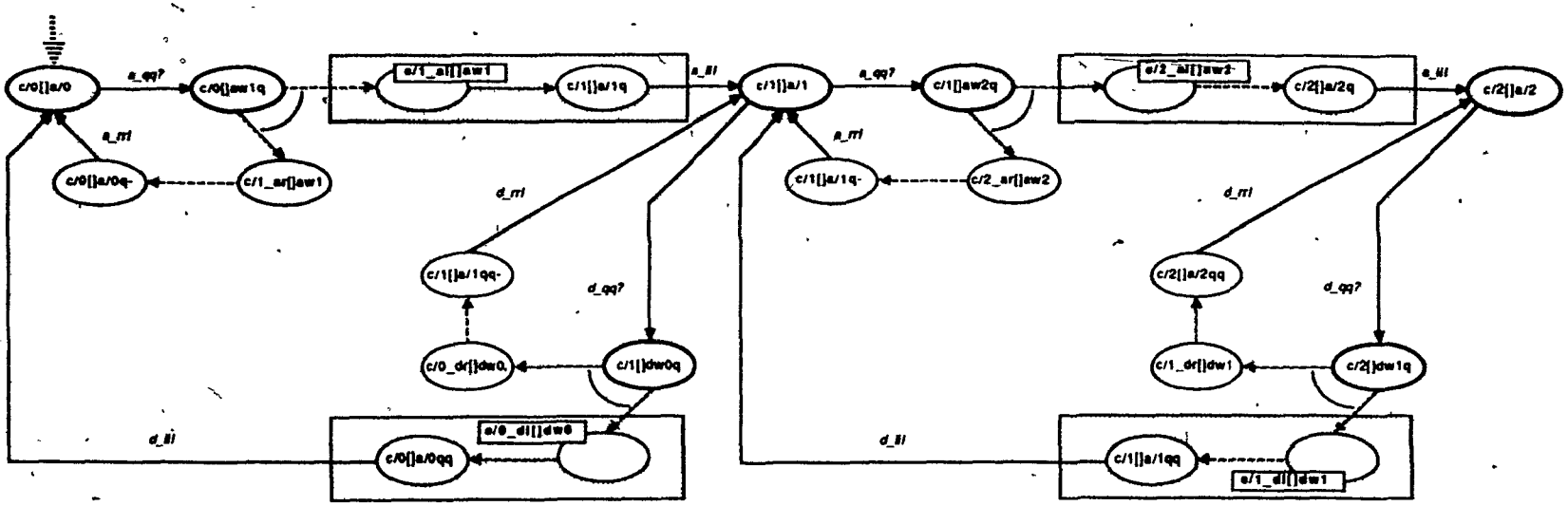
$$\mathtt{R(c/1(}g1.1\mathtt{)[]a/1qq(}x.1,y.1,n.1,m2.1\mathtt{))} \equiv \mathtt{R(c/1\_di(}x.1,y.1,n.1,g2.1\mathtt{)[]dw1(}x.1,y.1,n.1,m2.1\mathtt{))}$$

This simplifies the original system to 19 states (see Figure 4.7). Since the remaining internal events do not meet the stability criterion of Chapter 2, the system is said



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- see Appendix B for detailed process descriptions

Fig. 4.6 State-Transition Diagram for R for Single-Conference Membership-Based System



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- boxed pairs of states are equivalent (the state with name in bold replaces other one)
- see Appendix B for detailed process descriptions

Fig. 4.7 State-Transition Diagram for R-reduced for Single-Conference Membership-Based System

to be unstable.

Finally, if the *reason code* parameter sets involved in rejection procedures were collapsed to have cardinalities of one, the R-equivalence would hold for the four pairs of states:

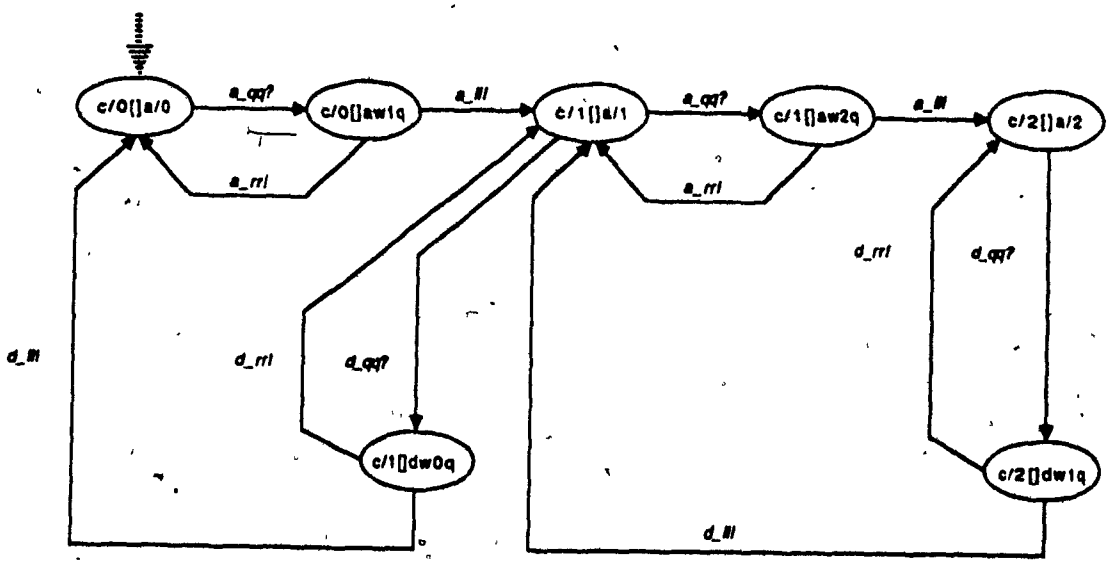
$$\texttt{R(c/0[]a/0q-)} \equiv \texttt{R(c/1\_ar[]aw1)}$$
$$\texttt{R(c/1[]a/1q-)} \equiv \texttt{R(c/2\_ar[]aw2)}$$
$$\texttt{R(c/1[]a/1qq-)} \equiv \texttt{R(c/0\_dr[]dw0)}$$
$$\texttt{R(c/1[]a/2qq-)} \equiv \texttt{R(c/1\_dr[]dw1)}$$

This would remove another source of instability. Figure 4.8 gives the *constrained* externally visible behaviour of this conferencing system. It remains an unstable system due to the four deterministic pairings for indication and rejection.

The combined processing for the []-product, R, equivalence tests, and reductions, using the automated DCP tools, took 4:58 minutes of CPU time. Page faults were 71.1k.

4.3 Study 2: Two-Conference System

The two-conference MBS formulation consists of the bridge in communication with two agents. Each agent has capacity for one member agent in addition to itself. Computer processing capability restricts this analysis to the []-product.



Notes: - parameters have been dropped from state names for ease of presentation
- see Appendix B for detailed process descriptions

Fig. 4.8 State-Transition Diagram for Constrained Single-Conference Membership-Based System

As an experiment, the []-product was first taken with the bridge kept to a maximum capacity of two agents (regardless of grouping). This ensues in the single deadlock, $c/2(g2.1)$ []aw2q$(x.1, y.1, n.1, m1.1)$ []aw2q$(x.2, y.2, n.2, m1.2)$, which had been anticipated. This deadlock is due to the situation where both agent processes have initiated conferences, with themselves as the sole participants thus far. Both then attempt to add a second participant to their respective conferences. At this point, however, the bridge can only accept a *delete* message from either agent, since it has reached its maximum capacity of two agents. If the bridge capacity was increased to handle four agents, it would be possible to create two conferences with two member agents each (which would have all three processes at maximum capacity), and eliminate the deadlock.

The []-product for the Two-Conference System, with the bridge returned to the maximum capacity of four agents, eliminates the aforementioned deadlock and yields 465 states in total. This is the same number of states found for the Two-Conference System of Chapter 3. Processing time was 58:04 minutes; page faults were in the vicinity of 804.7k.

The capacity of the bridge and agent processes can be arbitrarily extended to produce larger conferencing systems, with no material difference in the inter-operation of the processes.

Chapter 5 The Conference-Based System

The Conference-Based System (CBS) is almost identical in formulation to the MBS, being an evolutionary step. Behaviourally, however, the CBS formulation allows the bridge to operate on the basis of the number of outstanding conferences, regardless of the number of members in each conference, whereas the MBS-bridge operates by *counting* the total number of member agents registered in all conferences combined. The CBS approach thus ushers in a more concise description of the bridge and a new, economical *conference-kill* feature. With the incorporation of a new parameter, the now-traditional pair of studies, the Single-Conference System of a bridge and one agent and, the Two-Conference System of the same bridge with two agents, are reviewed.

5.1 Formulation of Processes

The natures of the CBS-bridge and CBS-agent are of the same spirit as the processes in the PS and MBS formulations. The bridge's capacity is now expressed in terms of number of conferences rather than agents. The present bridge may handle

two conferences each of arbitrarily large membership. The agent still supports a maximum of two agents (including itself) in a single conference. Only one agent or one conference is added to or deleted from the profile of the conference system with an appropriate *request* message. Both the bridge and agent process are deterministic.

The CBS message format is as described in Chapter 4 and illustrated in Figure 4.1.

5.1.1 Use of Parameters

The CBS formulation brings forth the new *test condition* parameter. The *target*, *requestor*, *conference name*, *membership list* and *reason code* parameters continue in their duties. The *grouping table* parameter is adjusted to *count* conferences rather than agents.

The CBS-bridge must distinguish between the addition or deletion of a conference, and the addition or deletion of a member agent from an existing conference. Of these four functions, all of which are specified and interpreted via the *target*, *requestor*, and *conference name* parameters, the route for the deletion of a conference is altered. The MBS formulation implicitly required the conference *chairman* to be removed last, at which point the conference would also be cleared. The CBS formulation does not restrict the ordering for removal of the *chairman*. The *chairman* may withdraw at any point from the conference, and through its withdrawal, *kill* the conference. This gives the flexibility of clearing a conference in its entirety, regardless of the number

of conferees, with a single, suitably-specified request - a very awkward task with the MBS-bridge, necessitating N *delete* requests for a conference of N agents (including and ending with the *chairman*).

The *test condition* or *tn.1* parameter is introduced in the bridge, as a means of correctly *directing* the []-product process, by appropriate conditional branching, to set apart conference and agent manipulation. Without such a device, the []-product would incur deadlocks, particularly in the Two-Conference System. Each set Tn contains a single and unique test. The test itself, in the form of a literal with conjunctions, disjunctions, etc., can be used as a *test condition* parameter name taking the place of Tn. Due to the binding procedures involved in creating the []-product, the *test condition* parameters must also appear and be matched in the agent's process description for the branching to operate.

It must be noted that this methodology for conditional branching is not entirely satisfactory. Although given the present work and the constraints of the automated tools used, it is effective and compact, the necessity for symmetric appearance of the *test condition* parameters in both processes is an artificial construction, and for increasingly complicated systems, prone to erroneous or cluttered results. Further, the cases underlying the branching must be conceived and verified by the protocol designer. As a more structured alternative, a prototype tool has recently been developed which builds sub-cases for each pair of parameters within a transition on the basis of an equality operator. This approach leads to an explicitly enumerated set

of sub-cases upon which branchings may be imposed, taking away the task of case construction from the designer. The drawback to this methodology is loss of compactness. It is anticipated that some hybrid of the two branching methodologies may be advantageous.

The *grouping table* parameter is characterized differently from the MBS formulation for the CBS-bridge, but is still responsible for tracking conferences by maintaining multiple membership lists with respective conference names and initiator (*chairman*) identities. Recalling (but modifying) the example of Chapter 4, if the parameter stands at $g2.1$, it now means that there are currently two conferences being tracked by the bridge. Externally, the number of member agents in each conference is not apparent, except by observation of the states of initiating agents. The parameter only proceeds to $g3.1$ when a new conference is created (in the presence of condition $t3$), or to $g1.1$ (with condition $t1$) when the *conference-kill* feature is invoked (equivalent to the removal of the *chairman*). Based on the request and the preceding grouping configuration two outcomes are possible. First, for the creation of conferences, a 1:1 mapping exists such that $gj{+}1.1$ is created given the parameters $gj.1$, $x.1$, $y.1$, and $n.1$. The inverse of this mapping must also be available for *conference-kill*. This mapping is simulated in the process descriptions by rewriting the preceding *grouping table* parameter by the current one, as before. Second, when an agent is added to or deleted from an existing conference, the table is updated internally, but the parameter name is left unaltered. Instead, another *table* is taken from the same set Gn which contains all n-conference *grouping tables* to manifest the change in membership.

5.1.2 Description of Agent

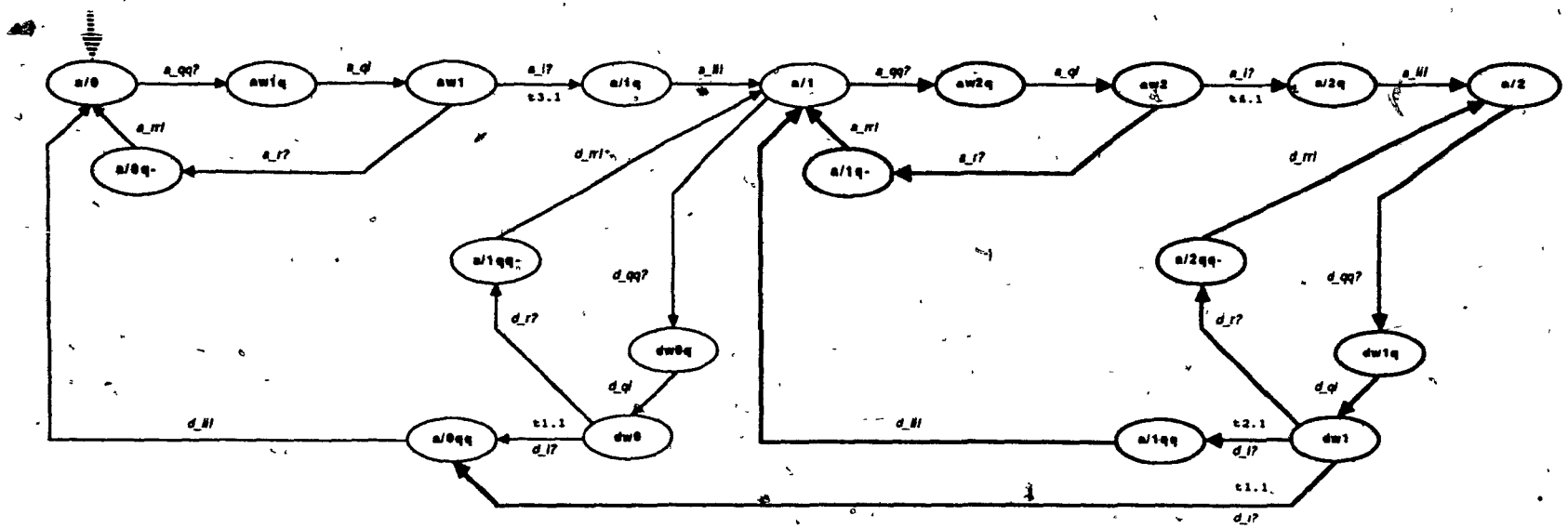
The state-transition diagram of 19 states for the CBS-agent is similar to the PS and MBS agent processes, with capacity for one member agent in addition to itself. The diagram is given in Figure 5.1 with the DCP expressions provided in Appendix C. The usage of messages and parameters is as for the MBS-agent.

The agent process, in contrast to previous formulations, now has the *conference-kill* facility. To operate, an extra transition must be inserted from `dw1` (and subsequently from every state `dwn`) to `a/0qq`, in order to collapse a conference from the agent's side. Concurrently, when a conference is collapsed, the *membership list* parameter in the agent falls from *m*n.1 to no list at all.

The CBS-agent can be replicated to form []-products of the bridge with N agent processes (N=1, 2 being the current examples), where the identities of agent processes are differentiated through the de-conflicting procedure of the tools. The formation of such a series of []-products is limited only by the conference capacity of the bridge, since each agent process, acting as a *chairman*, can generate a conference. This generic and modular agent process description is generalizable to any membership capacity, with the number of states equal to 9A+1 to handle A agents.

5.1.3 Description of Bridge

The state-transition diagram of the bridge process has 13 states and is drawn in Figure 5.2, with algebraic descriptions documented in Appendix C. The CBS-bridge



Notes - parameters have been dropped from state and message names for convenience
- bold lines outline the add/delete module per agent of additional capacity for a conference
- see Appendix C for detailed process descriptions

Fig. 5.1 State-Transition Diagram for Agent in Conference-Based System

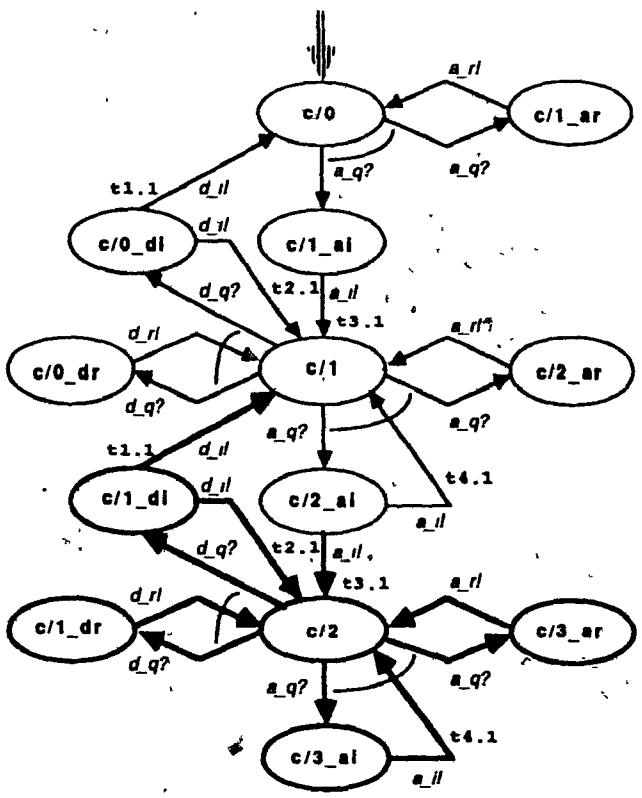
can support two conferences each of arbitrarily large membership. (The MBS-bridge was not limited to two conferences, but in the []-product with one or two agent processes, no more than two conferences could be created.) The bridge's decision-making capabilities remain in force (refer to flowchart of Figure 4.4).

The CBS-bridge has the major states, c/0, c/1, c/2, and has the same form as the single-conference PS-bridge or the MBS-bridge but for two extra states. c/0 is the ever-present empty state, where no conferences have as yet been constructed. The remaining major states indicate the total number of conferences, where the actual configuration is furnished within the *grouping table* parameter, *g*n.1. It is clear that this bridge process is again modular, as outlined in bold in Figure 5.2, and generalizable with 5C+3 states based on a capacity of C conferences.

In order to exemplify the operation of the CBS-bridge, state (8) of the process description is excerpted below:

```
(8) c/2(g2.1) =
      + #(x.1,y.1,n.1) .  a_q?(x.1,y.1,n.1) ; c/3_a1(x.1,y.1,n.1,g2.1)  --> (9)
      + #(x.1,y.1,n.1) .  a_q?(x.1,y.1,n.1) ; c/3_ar(x.1,y.1,n.1,g2.1)  --> (10)
      + #(x.1,y.1,n.1) .  d_q?(x.1,y.1,n.1) ; c/1_di(x.1,y.1,n.1,g2.1)  --> (11)
      + #(x.1,y.1,n.1) .  d_q?(x.1,y.1,n.1) ; c/1_dr(x.1,y.1,n.1,g2.1)  --> (12)
```

Here, the *state_name* c/2 indicates that two conferences exist with the total number of enrolled agents unavailable. All four transitions are of similar form, with the three parameters (*x*.1,*y*.1,*n*.1) drawn from *local_bindings*. This transition



Notes: - parameters have been dropped from state and message names for ease of presentation
- bold lines outline the add/delete module per conference of additional capacity
- see Appendix C for detailed process descriptions

Fig. 5.2 State-Transition Diagram for Bridge in Conference-Based System

equation, then, allows the bridge to accept one of two possible request messages, each of which has a pair of *deterministically selected* successor processes. The first pair of successors originate from an *add* request from a *requestor* $x.1$, with the target agent to be added $y.1$ under *conference name* $n.1$, all as in Chapter 4. The successor processes, `c/3_a1` and `c/3_ar`, are branchings generated from a bridge decision to accept or reject the request, as the suffixes suggest. Every one of the incoming parameter values, along with the original parameter $g2.1$, must be retained for the response. Once the response is emitted, only the $gn.1$ parameter is held for subsequent actions such as *delete* requests. The second pair of branchings arise from a *delete* request, carrying three parameter values as above. Again, branchings occur on success or failure of the request.

The bridge, in state (8), is at the maximum capacity of two conferences, which precludes the addition of more conferences but not more agents. State (9) returns to the major state `c/2`($g2.1$) or state (8) since only the $t4$ condition (to *add* an agent to an existing conference) may be satisfied. State (11) eventually goes to either major state `c/1`($g1.1$) or `c/2`($g2.1$), depending on whether the $t1$ (*delete* conference) or $t2$ (*delete* agent) condition is met, respectively. The two failures loop back to state (8).

5.2 Study 1: Single-Conference System

This study examines the bridge process in communication with a single agent process. Many similarities are found between this Single-Conference System and the

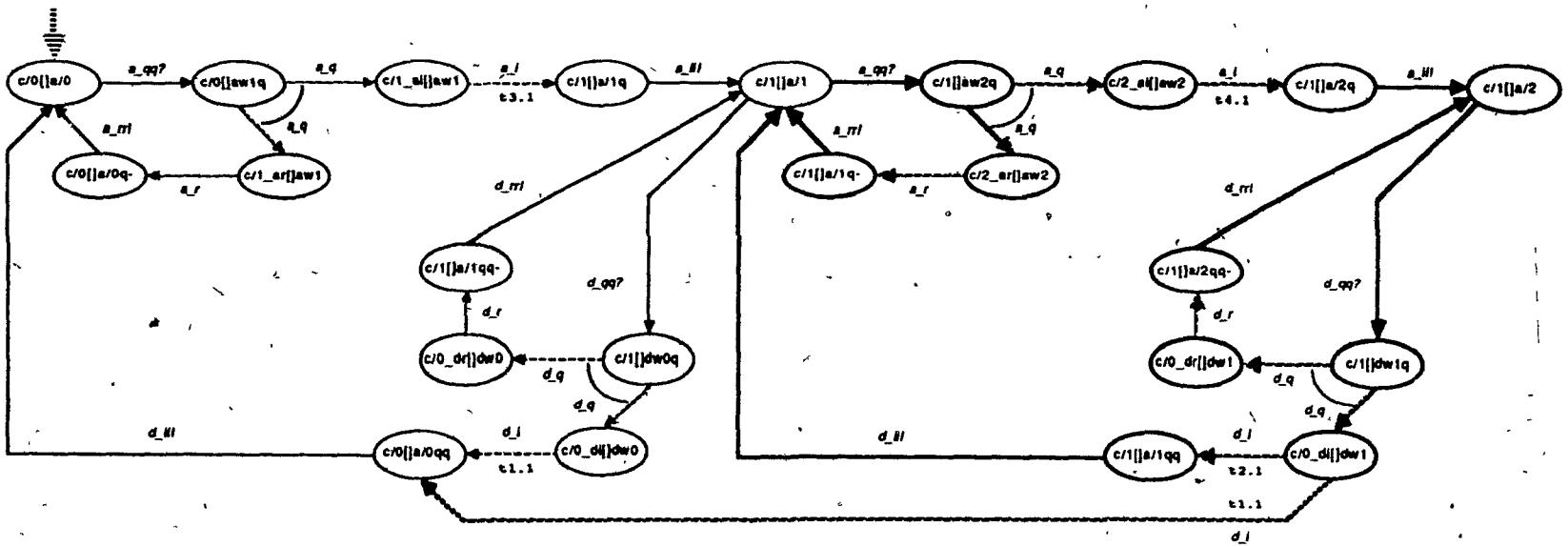
corresponding ones of preceding chapters.

The []-product is comprised of 23 states (the same number as the []-product of Study 1 in Chapters 3 and 4) and is deadlock-free. The graphical interpretation is given in Figure 5.3, while the algebraic output is available in Appendix C. Parameters are correctly carried and passed within and between the processes. The temporary storage by the bridge and requesting agent processes of all parameter values appearing in a request ensures error-free internal exchanges for responses, since a match must be achieved between the two sets of parameters (including the *test condition* parameter for branchings).

The modular structure of the []-product, as well as the inheritance of paired *deterministic selections* and the possibility of livelocks, are as observed in the PS and MBS Single-Conference Systems.

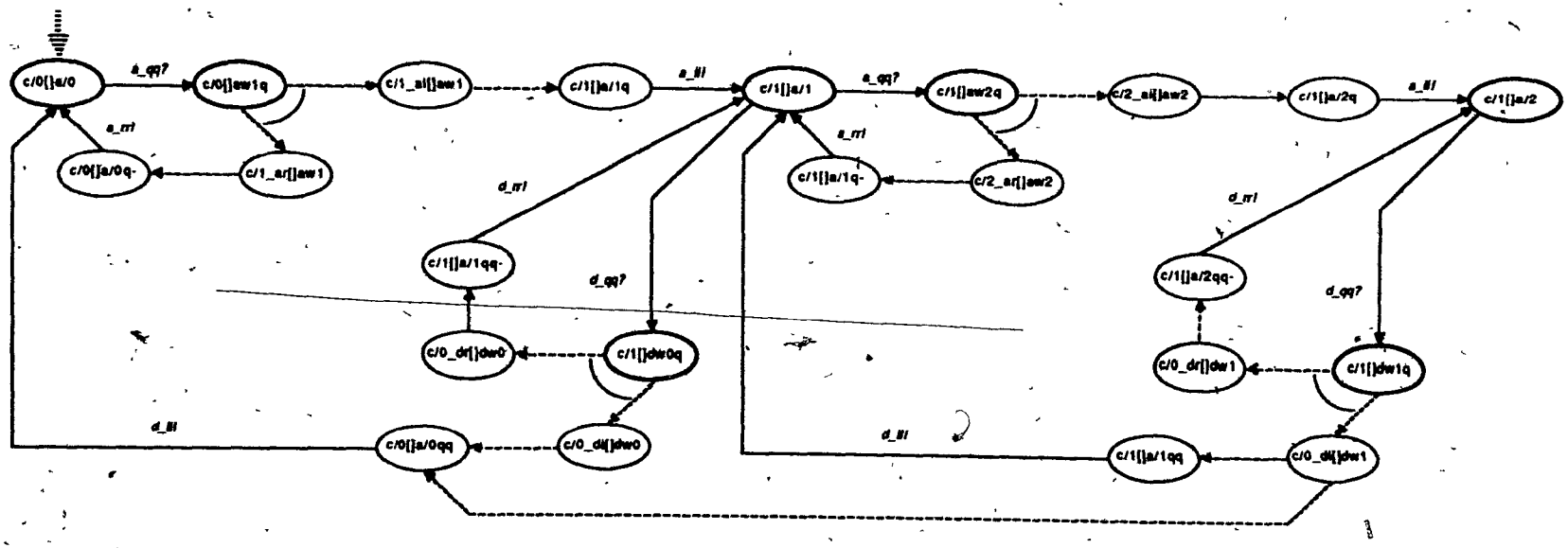
The application of the R-operator to the []-product yields a picture analogous to previous results, except for the behaviour at `c/0_di[]dw1`. `c/0_di[]dw1` has the two successor states `c/0[]a/0qq` and `c/1[]a/1qq` to allow for the *killing* of a conference or the deletion of an agent from a conference, as appropriate. The effect is rendered in Figure 5.4 and is provided in algebraic form in Appendix C.

Disposal of R-equivalent states would reduce the R of the conferencing system further. A set of equivalence assumptions were made by the tools and inductively



Notes
- parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- broken lines and single-letter suffixes in messages indicate internal events
- bold lines outline the add/delete module per agent of additional capacity for a conference
- see Appendix C for detailed process descriptions

Fig. 5.3 State-Transition Diagram for []-Product for Single-Conference Conference-Based System



Notes: - parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- see Appendix C for detailed process descriptions

Fig. 5.4 State-Transition Diagram for R for Single-Conference Conference-Based System

tested over four passes. This time, three equivalences were uncovered, these being:

$$\mathtt{R(c/1(}g1.1\mathtt{)[]a/1q(}x.1,y.1,n.1\mathtt{))} \equiv$$
$$\mathtt{R(c/1\_ai(}x.1,y.1,n.1\mathtt{)[]aw1(}x.1,y.1,n.1\mathtt{))}$$

$$\mathtt{R(c/1(}g1.1\mathtt{)[]a/2q(}x.1,y.1,n.1,m1.1\mathtt{))} \equiv$$
$$\mathtt{R(c/2\_ai(}x.1,y.1,n.1,g1.1\mathtt{)[]aw2(}x.1,y.1,n.1,m1.1\mathtt{))}$$

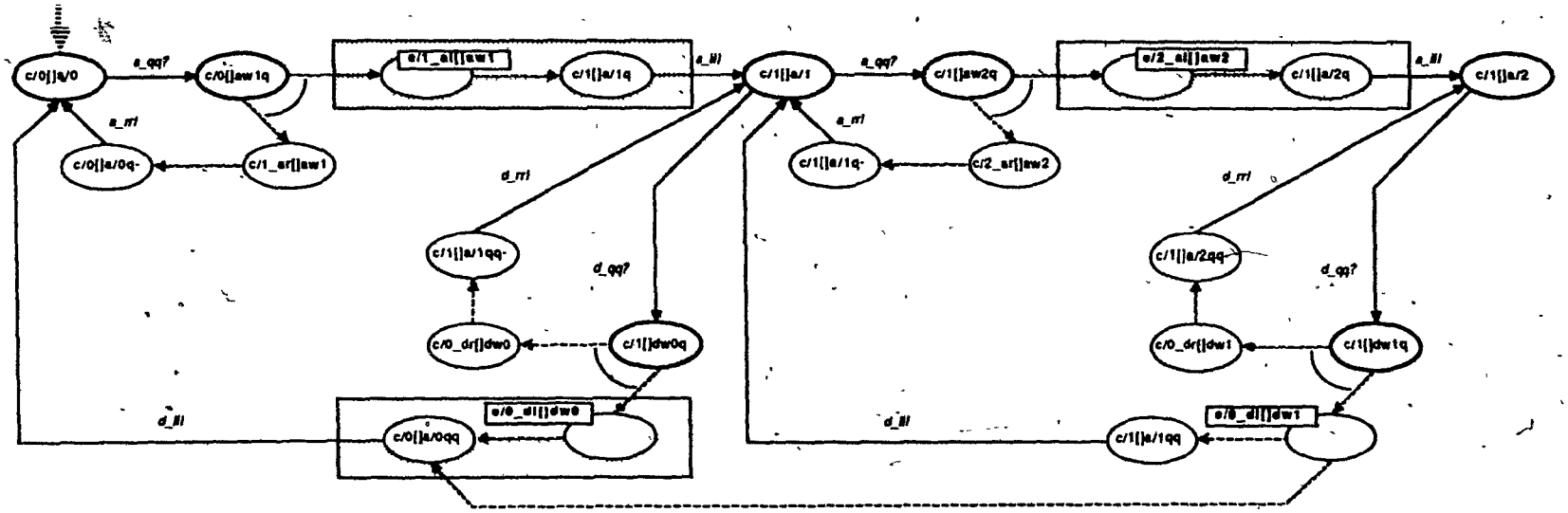
$$\mathtt{R(c/0[]a/0qq(}x.1,y.1,n.1\mathtt{))} \equiv$$
$$\mathtt{R(c/0\_di(}x.1,y.1,n.1,g1.1\mathtt{)[]dw0(}x.1,y.1,n.1,m1.1\mathtt{))}$$

This simplifies the original system to 20 states as shown in Figure 5.5. The extra transition from `c/0_di[]dw1` prevents the inclusion of the fourth equivalence of the previous formulations. The Conference-Based System is unstable by the definition of Chapter 2.

Finally, if the *reason code* parameter sets were obliged to have cardinalities of one, the R-equivalence would hold for the four pairs of states:

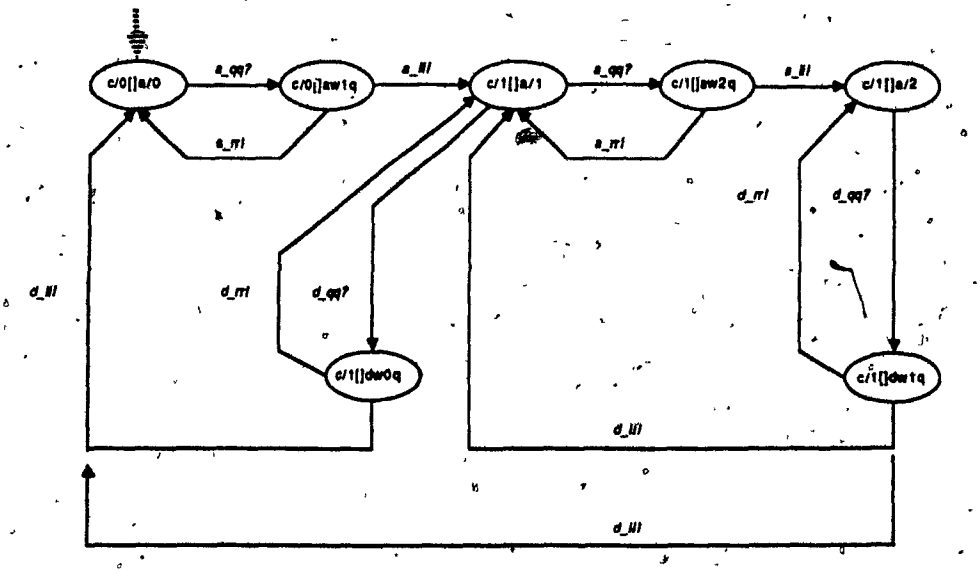
$$\mathtt{R(c/0[]a/0q-\ )} \equiv \mathtt{R(c/1\_ar[]aw1)}$$
$$\mathtt{R(c/1[]a/1q-\ )} \equiv \mathtt{R(c/2\_ar[]aw2)}$$
$$\mathtt{R(c/1[]a/1qq-)} \equiv \mathtt{R(c/0\_dr[]dw0)}$$
$$\mathtt{R(c/1[]a/2qq-)} \equiv \mathtt{R(c/1\_dr[]dw1)}$$

This would remove another source of instability. Figure 5.6 gives the *constrained* externally visible behaviour of this conferencing system. It remains an unstable system due to the four deterministic pairings for indication and rejection.



Notes
- parameters have been dropped from state and message names for ease of presentation
- arcs indicate paired deterministic selections
- solid lines and double-letter suffixes in messages indicate external events
- bold ellipses are externally visible states
- boxed pairs of states are equivalent (the state with name in bold replaces other one)
- see Appendix C for detailed process descriptions

Fig. 5.5 State-Transition Diagram for R-reduced for Single-Conference Conference-Based System



Notes: - parameters have been dropped from state and message names for ease of presentation
- see Appendix C for detailed process descriptions

Fig. 5.6 State-Transition Diagram for Constrained Single-Conference Conference-Based System.

The joint processing for the $\|$-product, R'equivalence tests, and reductions, using the automated DCP tools, required 6.02 minutes of CPU time. Page faults were 90.8k.

5.3 Study 2: Two-Conference System

The two-conference CBS formulation consists of the bridge in communication with two agents. Each agent has capacity for one member agent in addition to itself. Computer processing capability restricts this analysis to the $\|$-product.

The $\|$-product for the Two-Conference System produces 165 states in total. Interestingly, this is the same number of states found for the Two-Conference Systems of Chapters 3 and 4. Processing time amounted to 56.51 minutes, page faults were approximately 774.1k. Both figures are less than their counterparts in Study 2 of Chapter 4. Although an extra parameter and some additional transitions are present in this formulation, the computing gains are achieved due to the economical CBS bridge process. (In the Single-Conference System, the additional elements overshadow any prospect for gains.)

The capacity of the bridge and agent processes can be extended to produce larger conferencing systems. The above differences in compactness and computing would, of course, become more pronounced and increasingly in favour of the CBS formulation for bridges and agents of greater conference- and membership-handling capacity.

Chapter 6 Conclusions and Future Direction

This final chapter recalls the results of the three previous chapters, provides a set of general conclusions and gives some short comments on the possible future activities in this field.

6.1. Summary of Results

Table 6.1 lists the formulations and findings for the Preliminary-, Membership-Based-, and Conference-Based-Systems, as reported in Chapters 3, 4 and 5, respectively. Commencing at the top of the Table, several observations may be made:

- Reading from left to right, the use of parametrization increases, from two parameters in the PS formulation to seven in the MBS. With the formulations thus acquiring greater expressive power, the functions over which messages are based are reduced from four to two and, the carriage of agent identities in state- and message-names is eliminated.
- The agent process description, in all cases, has 19 states and can support two member agents (including itself). However, in the PS, the agent has a two-element memory (for the target agent and reason code), while in the MBS and CBS, it has a five-element memory (for the agent or requestor

	FORMULATION					
	PS		MBS		CBS	
	Single-Conf	Two-Conf	Single-Conf	Two Conf	Single-Conf	Two-Conf
Parameters	target, reason code		+ requestor, conference name, grouping table, membership list		+ test condition	
Message Architecture	create, add, delete, clear		add, delete			
Agent ID	in state- and message-names		parametrized			
Agent (states)	19					
Agent (general)	n/a		9A + 1			
Bridge (states)	11	57	21		13	
Bridge (general)	n/a		5A + 1		5C + 3	
[]-Product (states)	23	465	23	465	23	465
R-reduced (states)	19	n/a	19	n/a	20	n/a
Computation Time (mn)	1 18	20 39	4 58	58 04	6 02	56 51
Page Faults (k)	20 7	278 7	71.1	804 7	90 8	774.1

Table 6.1 Summary of Results

identity target agent, conference name, membership list, and reason code) In the CBS the agent is able to directly collapse a conference that it has initiated All three agent processes are generalizable with 9A+1 states to handle A agents

- The bridge process description as it is augmented with parametrization, changes from an inflexible one with different bridges necessary for the Single- and Two-Conference PS where each conference has capacity for two members, to a very compact, unified process with arbitrarily large membership capacity within each conference in the CBS The PS bridges have one element of memory for each target agent, the MBS and CBS bridges have four-element memories for the requestor, target agent, conference name, and grouping table. The MBS- and CBS-bridges can be generalized with 5A+1 states to handle A agents for the former, and 5C+3 states for C conferences for the latter Decison-making capabilities for the bridges are also modified in successive formulations.
- The Single-Conference []-product exhibits 23 states, without deadlocks, for all formulations The form of the []-product is also consistent across the three, with the exception of the extra transition for the *conference-kill* feature in the CBS []-product. The Two-Conference []-product yields 465 states, without deadlocks, for all three formulations.
- The Single-Conference R-reduced gives 19 states for the PS and MBS with four equivalences found, and 20 states for the CBS with three equivalences discovered, the latter due to the extra transition for *conference-kill*. The R-reduced is not performed for the Two-Conference systems.
- Computation time and page faults grow with increased complexity (left to right) for Single-Conference systems. For Two-Conference systems, there is an increase from the PS to the MBS, but a dip from the MBS to the CBS due to the very economical bridge description in the latter.

6.2 General Conclusions

It is evident from the evolution of and the resultant computations for the formulations of Chapters 3, 4 and 5 that:

- A foundation, including a conceptual framework for the *bridge* and *agent* discrete processes, as well as a notation for a message architecture and parameters, has been constructed for further exploration of conferencing,
- The utility and validity of parametrization has been investigated in the context of the current formulations, with progressively positive indications for compactness, process generalization, correctness, and computational efficiency in describing the bridges, agents, and their interactions, as highlighted by the Conference Based System,
- A branching mechanism, employing *test condition* parameters, has been used to substantial advantage, providing insight toward the development of branching and case constructs, and,
- The usefulness of automated protocol analysis tools, particularly those based on DCP, has been demonstrated

6.3 Future Work

Future work in protocols for conferencing systems should proceed so as to:

- Extend the present formulations to incorporate probes to and acknowledgments from *target* agents, multi-channel operation, various error-detection and error-survival schemes in information transfer, enhanced features for more advanced conference manipulation, and bridge-bridge communication for multi-bridge scenarios;
- Construct an algebra on the present untyped sets to provide for functions of parameters and relations (beyond the = comparator) among parameters; and,
- Further develop the automated tools, specifically for branching and subcase analysis, as well as for inclusion of synthesis procedures.

References

1. G.B. Thompson, "The Challenge of Choice", *Intermedia*, Vol 12, No 4/5, pp. 60-63, July/Sept., 1984

2. D. McConnell, "The Impact of Cyclops Shared-Screen Teleconferencing in Distance Education", *British Journal of Educational Technology*, Vol 17, No. 1, pp. 41-74, Jan., 1986.

3. M.D. Benjamin, R.D Johnston and B. Prasada, "A Model for Computer-Mediated Interactive Visual Communications", *Proc International Conference on Communications*, Boston, MA, pp 56.2.1-56.2.6, June 10-14, 1979.

4. H.M. Lipinski and R.H. Miller, "FORUM: A Computer-Assisted Communications Medium", *Proc. Second International Conference on Computer Communications*, pp. 143-147, August, 1974.

5. W. Pferd, L.A. Peralta and F.X. Prendergast, "Interactive Graphics Teleconferencing", *IEEE Computer*, Vol. 12, No. 11, pp. 62-72, Nov., 1979.

6. C.W. Kelly III, "An Enhanced Presence Video Teleconferencing System", *Proc. CompCon*, Washington, DC, pp. 544-551, Sept. 20-23, 1982.

7. S. Randall, "The Shared Graphic Workspace: Interactive Data Sharing in a Teleconference Environment", *Proc. CompCon*, IEEE Computer Society, Los Alamitos, CA, pp. 535-542, Fall, 1982.

8. H. Lipinski and R. Adler, "Electronic Communication for Interactive Group Modelling", in *Computer Networks and Simulation II*, S. Schoemaker (ed.), North Holland, New York, pp. 251-277, 1982.

9. A. Burns and M.A. Rathwell, "A Communications Environment for Co-operative Information Systems Development", *Software Engineering Journal*, Vol. 2, No. 1, pp. 9-14, Jan., 1987.

10. L. Aguilar, "A Format for a Graphical Communications Protocol", *IEEE Computer Graphics and Applications*, Vol. 6, No. 2, pp. 52-62, April, 1980.

11. N. Meyrowitz and A. van Dam, "Interactive Editing Systems (Parts I and II)", *ACM Computing Surveys*, Vol. 14, No. 3, pp. 321-415, Sept., 1982.

12. K.S. Sarin, "Interactive On-Line Conferences", PhD. dissertation, MIT, Dept. of Electrical Engineering and Computer Science, 1984.

13. K. Rea, "Automating the Analysis and Design of Discrete Communicating Processes", M.Eng. Thesis, Dept. of Electrical Engineering, McGill University, August, 1984.

14. P.V. Mockapetris, "Analysis of Reliable Multicast Algorithms for Local Networks", *Proc. Ninth Data Communications Symposium*, pp. 150-157, Oct., 1983.

15. R. Pardo and M.T. Liu, "Multidestination Protocols for Distributed Systems", *Proc. Computer Networking Symposium*, National Bureau of Standards, Gaithesburg, Maryland, pp. 176-185, 1979.

16. K.P. Birman and T.A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transactions on Computer Systems*, Vol. 5, No. 1, pp. 47-76, Feb., 1987.

17. K. Fung, J. Luetchford and I. Scales, "ISDN Standards Issues", *Telesis*, Vol. 13, No. 3, pp. 25-33, 1986.

18. J. Chatterley, B. Newman and R. Wellard, "A Fuzzy Concept Takes Some Shape", *Computing Canada*, pp. 11-12, August 21, 1986.

19. International Organization for Standardization, "Information Processing Systems - Open Systems Interconnection - Basic Reference Model", Ref No ISO 7498-1984(E), Oct., 1984.

20. J.D. Day and H. Zimmermann, "The OSI Reference Model", *Proc. IEEE*, Vol. 71, No. 12, pp. 1334-1340, Dec., 1983.

21. C.A. Sunshine, "Survey of Protocol Definition and Verification Techniques", *Computer Networks*, Vol. 2, pp. 346-350, Oct., 1978.

22. R.L. Schwartz and P.M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards", *IEEE Transactions on Communications*, Vol. COM-30, No. 12, pp. 2486-2496, Dec., 1982.

23. A.A.S. Danthine, "Protocol Representation with Finite-State Models", *IEEE Transactions on Communications*, Vol. COM-28, No. 4, pp. 632-643, April, 1980.

24. P.M. Merlin, "Specification and Validation of Protocols", *IEEE Transactions on Communications*, Vol. COM-27, No. 11, pp. 1671-1680, Nov., 1979.

25. J.L. Peterson, "Petri Net Theory and The Modelling of Systems", Prentice-Hall Inc., Englewood Hills, NJ, 1981.

26. S.W. Lai, E.D. Brown and S.E. Tavares, "Modelling Communication Protocols with Coloured-Arc Petri Nets", *Proc. Thirteenth Biennial Symposium on Communications*, Queen's University, Kingston, ONT, pp. D.3.9-D.3.12, June 2-4, 1986.

27. M. Diaz, "Modelling and Analysis of Communication and Cooperation Protocols Using Petri-Net Based Models", *Proc. Second International Workshop on Protocol Specification, Testing and Verification*, May, 1982.

28. A.S. Tanenbaum, "Computer Networks", Prentice-Hall Inc., Englewood Hills, NJ,

1981

29 C A Vissers, R.L Tenney and G.V Bochmann, "Formal Description Techniques", *Proc IEEE*, Vol 71, No. 12, pp 1356 1364, Dec , 1983

30 K. Kurosawa, H. Koike and S Tsuji, "A New Specification and Validation Method for Communication Protocols - A Proposal of a Composite State Diagram Mixed with Logical Relations", *Proc IEEE Global Telecommunications Conference*, Vol. 1, Atlanta, GA, pp 6.5.1-6 5.7, Nov 26-29, 1984

31. U. Pletat, "Algebraic Specifications of Abstract Data Types and CCS. An Operational Junction", *Proc Sixth International Workshop on Protocol Specification, Testing and Verification*, Montreal, Canada, pp 10.13-10.21, June 10-13, 1986

32. G.V. Bochmann and C A. Sunshine, "Formal Methods in Communication Protocol Design", *IEEE Transactions on Communications*, Vol. COM-28, No 4, pp. 624-631, April, 1980.

33. G. Berthelot and R. Terrat, "Petri Nets Theory for the Correctness of Protocols", *IEEE Transactions on Communications*, Vol. COM-30, No. 12, pp. 2497-2505, Dec., 1982.

34. R.E. Miller and G.M. Lundy, "An Approach to Modelling Communication Protocols Using Finite State Machines and Shared Variables", *IEEE Global Telecommunications Conference*, Houston, TX, pp. 3.8.1-3.8.5, Dec. 1-4, 1986.

35. R.deB. Johnston, "A Mathematical Model of Discrete Communication Processes", *INRS-Télécommunications Technical Report*, May, 1985.

36. R. Milner, "A Calculus of Communicating Systems", Springer-Verlag, Berlin, 1980.

37. C.A.R. Hoare, "Communicating Sequential Processes", *Communications of the*

ACM Vol 21, No. 8, pp 666-677 August, 1978

38. M. Wand, "Induction, Recursion and Programming", North Holland, New York, 1980

39. P.A. Cloutier, "Évaluation d'un Systeme Automatisé de Vérification de Protocoles", M.Eng. Thesis, INRS-Télécommunications, in progress.

40. P. Halmós, "Naive Set Theory", Van Nostrand Inc., New York, 1960, p. 93.

41. Z. Manna, "Mathematical Theory of Computation", McGraw-Hill Co., New York, 1974.

42. T A. Nguyen, "Introducing Parameterized State/Transition Descriptions into Communicating Processes", M.Eng. Thesis, Dept. of Electrical Engineering, McGill University, in progress.

Appendix A. DCP Expressions and Results for Preliminary System

BRIDGE (Single-Conference)

NB. The following correspondences hold for state names in Appendix A and text of Chapter 3:

i4, r4 = i2, r2 ; i5, r5 = i3, r3 ; i13, r13 = i4, r4 ; c/2_1 = c/1_2

(0) c/0 =

+ 1c_q? ; i1 --> (1)

+ 1c_q? ; r1 --> (2)

(1) i1 =

+ 1c_i! ; c/1_1 --> (3)

(2) r1 =

+ #(c.1) . 1c_r!(c.1) ; c/0 --> (0)

(3) c/1_1 =

+ 1l_q? ; i4 --> (4)

+ 1l_q? , r4 --> (5)

+ #(x.1) . 1a_q?(x.1) ; i5(x.1) --> (6)

+ #(x.1) . 1a_q?(x.1) ; r5(x.1) --> (7)

(4) i4 =

+ 1l_i! ; c/0 --> (0)

(5) r4 =

+ #(l.1) . 1l_r!(l.1) ; c/1_1 --> (3)

(6) i5(x.1) =

+ 1a_i!(x.1) ; c/2_1(x.1) --> (8)

(7) r5(x.1) =

+ #(a.1) . 1a_r!(x.1,a.1) ; c/1_1 --> (3)

(8) c/2_1(x.1) =

+ 1d_q?(x.1) ; i13(x.1) --> (9)

+ 1d_q?(x.1) ; r13(x.1) --> (10)

(9) i13(x.1) =

+ 1d_i!(x.1) ; c/1_1 --> (3)

(10) r13(x.1) =

+ ∑(d.1) . 1d_r!(x.1,d.1) ; c/2_1(x.1) --> (8)

AGENT1 ----------

(0) 1a/0 =

+ 1c_qq? ; 1aw1q --> (1)

(1) 1aw1q =

+ 1c_q! ; 1aw1 --> (2)

(2) 1aw1 =

+ 1c_i? ; 1a/1q --> (3)

+ ∑(c.1) . 1c_r?(c.1) ; 1a/0q(c.1) --> (4)

(3) 1a/1q =

+ 1c_ii! ; 1a/1 --> (5)

(4) 1a/0q(c.1) =

+ 1c_rr!(c.1) ; 1a/0 --> (0)

(5) 1a/1 =

+ ∑(x 1) . 1a_qq?(x.1) ; 1aw2q(x.1) --> (6)

+ 1!_qq? ; 1dw0q --> (7)

(6) 1aw2q(x.1) =

+ 1a_q!(x.1) ; 1aw2(x.1) --> (8)

(7) 1dw0q =

+ 1!_q! ; 1dw0 --> (9)

(8) 1aw2(x.1) =

```
        + 1a_i?(x.1) , 1a/2q(x.1)      --> (10)
        + #(a 1)   1a_r?(x 1,a 1) ; 1a/1qqq(x 1,a 1)      --> (11)

(9)  1dw0  =
        + 1l_i? , 1a/0qq      --> (12)
        + #(l.1) . 1l_r?(l 1) , 1a/1qq(l 1)      --> (13)

(10)  1a/2q(x.1)  =
        + 1a_ii'(x.1) ; 1a/2(x 1)      --> (14)

(11)  1a/1qqq(x.1,a 1)  =
        + 1a_rr'(x 1,a 1) , 1a/1      --> (5)

(12)  1a/0qq  =
        + 1l_ii'; 1a/0      --> (0)

(13)  1a/1qq(l 1)  =
        + 1l_rr'(l 1) ; 1a/1      --> (5)

(14)  1a/2(x.1)  =
        + 1d_qq?(x.1) ; 1dw1q(x.1)      --> (15)

(15)  1dw1q(x.1)  =
        + 1d_q'(x.1) ; 1dw1(x 1)      --> (16)

(16)  1dw1(x.1)  =
        + 1d_i?(x.1) ; 1a/1qq(x.1)      --> (17)
        + #(d 1) . 1d_r?(x 1,d 1) ; 1a/2qq(x 1,d 1)      --> (18)

(17)  1a/1qq(x 1)  =
        + 1d_ii'(x 1) , 1a/1      --> (5)

(18)  1a/2qq(x.1,d.1)  =
        + 1d_rr'(x 1,d 1) , 1a/2(x.1)      --> (14)

BRIDGE [] AGENT1 ----------------------------------------------------------------
```

(0) (c/0 [] la/0) =

  + lc_qq? ; (c/0 [] law1q) --> (1)

(1) (c/0 [] law1q) =

  + lc_q , (i1 [] law1) --> (2)

  + lc_q ; (r1 [] law1) --> (3)

(2) (i1 [] law1) =

  + lc_i , (c/1_1 [] la/1q) --> (4)

(3) (r1 [] law1) =

  + #(c 1) lc_r(c 1) , (c/0 [] la/0q(c 1)) --> (5)

(4) (c/1_1 [] la/1q) =

  + lc_ii ; (c/1_1 [] la/1) --> (6)

(5) (c/0 [] la/0q(c 1)) =

  + lc_rr'(c.1) , (c/0 [] la/0) --> (0)

(6) (c/1_1 [] la/1) =

  + #(x 1) . la_qq?(x 1) , (c/1_1 [] law2q(x 1)) --> (7)

  + 1!_qq? , (c/1_1 [] ldw0q) --> (8)

(7) (c/1_1 [] law2q(x 1)) =

  + la_q(x 1) , (i5(x 1) [] law2(x 1)) --> (9)

  + la_q(x.1) , (r5(x 1) [] law2(x 1)) --> (10)

(8) (c/1_1 [] ldw0q) =

  + 1!_q , (i4 [] ldw0) --> (11)

  + 1!_q , (r4 [] ldw0) --> (12)

(9) (i5(x 1) [] law2(x 1)) =

  + la_i(x 1) , (c/2_1(x 1) [] la/2q(x 1)) --> (13)

(10) (r5(x 1) [] law2(x 1)) =

```
        + #(a 1)   1a_r(x.1,a.1) , (c/1_1 [] 1a/1qqq(x 1,a 1))      --> ( 14)

( 11)  (i4 [] 1dw0) =
        + 1l_i ; (c/0 [] 1a/0qq)      --> ( 15)

( 12)  (r4 [] 1dw0) =
        + #(l 1)   1l_r(l 1) , (c/1_1 [] 1a/1qq(l 1))      --> ( 16)

( 13)  (c/2_1(x.1) [] 1a/2q(x.1)) =
        + 1a_i1'(x 1) , (c/2_1(x 1) [] 1a/2(x 1))      --> ( 17)

( 14)  (c/1_1 [] 1a/1qqq(x 1,a 1)) =
        + 1a_rr'(x 1,a.1) ; (c/1_1 [] 1a/1)      --> ( 6)

( 15)  (c/0 [] 1a/0qq) =
        + 1l_i1' , (c/0 [] 1a/0)      --> ( 0)

( 16)  (c/1_1 [] 1a/1qq(l 1)) =
        + 1l_rr'(l.1) , (c/1_1 [] 1a/1)      --> ( 6)

( 17)  (c/2_1(x 1) [] 1a/2(x 1)) =
        + 1d_qq?(x.1) ; (c/2_1(x.1) [] 1dw1q(x.1))      --> ( 18)

( 18)  (c/2_1(x 1) [] 1dw1q(x 1)) =
        + 1d_q(x.1) , (i13(x 1) [] 1dw1(x 1))      --> ( 19)
        + 1d_q(x 1) ; (r13(x.1) [] 1dw1(x 1))      --> ( 20)

(.19)  (i13(x 1) [] 1dw1(x.1)) =
        + 1d_i(x 1) ; (c/1_1 [] 1a/1qq(x 1))      --> ( 21)

( 20)  (r13(x 1) [] 1dw1(x.1)) =
        + #(d.1) . 1d_r(x.1,d.1) ; (c/2_1(x.1) [] 1a/2qq(x 1,d.1))      --> ( 22)

( 21)  (c/1_1 [] 1a/1qq(x 1)) =
        + 1d_i1'(x 1) ; (c/1_1 [] 1a/1)      --> ( 6)
```

(22) (c/2_1(x 1) [] 1a/2qq(x 1,d 1)) =

  + 1d_rr'(x.1,d 1) ; (c/2_1(x 1) [] 1a/2(x 1)) --> (17)

R (BRIDGE [] AGENT1) ----------

(0) R((c/0 [] 1a/0)) =

  + 1c_qq? , R((c/0 [] 1aw1q)) --> (1)

(1) R((c/0 [] 1aw1q)) =

  + 1c_i1' , R((c/1_1 [] 1a/1)) --> (6)

  + #(c.1) . 1c_rr'(c 1) ; R((c/0 [] 1a/0)) --> (0)

(2) R((i1 [] 1aw1)) =

  + 1c_i1' ; R((c/1_1 [] 1a/1)) --> (6)

(3) R((r1 [] 1aw1)) =

  + #(c 1) . 1c_rr'(c.1) , R((c/0 [] 1a/0)) --> (0)

(4) R((c/1_1 [] 1a/1q)) =

  + 1c_i1' ; R((c/1_1 [] 1a/1)) --> (6)

(5) R((c/0 [] 1a/0q(c 1))) =

  + 1c_rr'(c 1) , R((c/0 [] 1a/0)) --> (0)

(6) R((c/1_1 [] 1a/1)) =

  + #(x 1) 1a_qq?(x 1) , R((c/1_1 [] 1aw2q(x 1))) --> (7)

  + 1i_qq? ; R((c/1_1 [] 1dw0q)) --> (8)

(7) R((c/1_1 [] 1aw2q(x 1))) =

  + 1a_i1'(x.1) , R((c/2_1(x 1) [] 1a/2(x 1))) --> (17)

  + #(a.1) . 1a_rr'(x 1,a 1) ; R((c/1_1 [] 1a/1)) --> (6)

(8) R((c/1_1 [] 1dw0q)) =

  + 1i_i1' ; R((c/0 [] 1a/0)) --> (0)

  + #(i.1) 1i_rr'(i 1) , R((c/1_1 [] 1a/1)) --> (6)

```
( 9)  R((i5(x 1) [] 1aw2(x 1))) =
        + 1a_ii'(x.1) ; R((c/2_1(x.1) [] 1a/2(x 1)))      --> ( 17)

( 10)  R((r5(x.1) [] 1aw2(x.1))) =
        + #(a.1) . 1a_rr'(x.1,a.1) , R((c/1_1 [] 1a/1))      --> ( 6)

( 11)  R((i4 [] 1dw0)) =
        + 1l_ii' ; R((c/0 [] 1a/0))      --> ( 0)

( 12)  R((r4 [] 1dw0)) =
        + #(l 1) . 1l_rr'(l 1) ; R((c/1_1 [] 1a/1))      --> ( 6)

( 13)  R((c/2_1(x.1) [] 1a/2q(x 1))) =
        + 1a_ii'(x.1) ; R((c/2_1(x 1) [] 1a/2(x.1)))      --> ( 17)

( 14)  R((c/1_1 [] 1a/1qqq(x 1,a.1))) =
        + 1a_rr'(x.1,a.1) ; R((c/1_1 [] 1a/1))      --> ( 6)

( 15)  R((c/0 [] 1a/0qq)) =
        + 1l_ii' ; R((c/0 [] 1a/0))      --> ( 0)

( 16)  R((c/1_1 [] 1a/1qq(l.1))) =
        + 1l_rr'(l 1) , R((c/1_1 [] 1a/1))      --> ( 6)

( 17)  R((c/2_1(x 1) [] 1a/2(x 1))) =
        + 1d_qq?(x 1) , R((c/2_1(x.1) [] 1dw1q(x.1)))      --> ( 18)

( 18)  R((c/2_1(x 1) [] 1dw1q(x 1))) =
        + 1d_ii'(x 1) , R((c/1_1 [] 1a/1))      --> ( 6)
        + #(d.1) . 1d_rr'(x.1,d.1) ; R((c/2_1(x.1) [] 1a/2(x.1)))      --> ( 17)

( 19)  R((i13(x.1) [] 1dw1(x.1))) =
        + 1d_ii'(x.1) ; R((c/1_1 [] 1a/1))      --> ( 6)

( 20)  R((r13(x 1) [] 1dw1(x 1))) =
```

```
        + #(d.1) . 1d_rr'(x.1,d.1) , R((c/2_1(x 1) [] 1a/2(x 1)))      --> ( 17)

( 21)  R((c/1_1 [] 1a/1qq(x 1))) =

        + 1d_ii'(x.1) ; R((c/1_1 [] 1a/1))      --> ( 6)

( 22)  R((c/2_1(x.1) [] 1a/2qq(x.1,d.1))) =

        + 1d_rr'(x.1,d 1) , R((c/2_1(x.1) [] 1a/2(x 1)))      --> ( 17)

Starting  pass 1

Starting  pass 2

EQUIVALENCES ------------------------------------------------------------------

== (R((c/1_1 [] 1a/1q)) , R((i1 [] 1aw1)))
    -->   (4) == (2)
== (R((c/2_1(x.1) [] 1a/2q(x.1))) , R((i5(x 1) [] 1aw2(x.1))))
    -->   (13) == (9)
== (R((c/0 [] 1a/0qq)) , R((i4 [] 1dw0)))
    -->   (15) == (11)
== (R((c/1_1 [] 1a/1qq(x.1))) , R((i13(x 1) [] 1dw1(x 1))))
    -->   (21) == (19)

R-REDUCED ---------------------------------------------------------------------

 (0)  R((c/0 [] 1a/0))  =

       + 1c_qq' ; R((c/0 [] 1aw1q))      --> (1)

 (1)  R((c/0 [] 1aw1q))  =

       + 1c_ii' , R((c/1_1 [] 1a/1))      --> (5)

       + #(c.1)   1c_rr'(c 1) ; R((c/0 [] 1a/0))      --> (0)

 (2)  R((i1 [] 1aw1))  =

       + 1c_ii' , R((c/1_1 [] 1a/1))      --> (5)

 (3)  R((i1 [] 1aw1))  =

       + #(c.1)   1c_rr'(c 1) ; R((c/0 [] 1a/0))      --> (0)

 (4)  R((c/0 [] 1a/0q(c 1)))  =

       + 1c_rr'(c.1) ; R((c/0 [] 1a/0))      --> (0)

 (5)  R((c/1_1 [] 1a/1))  =
```

```
      + #(x.1) . 1a_qq?(x.1) ; R((c/1_1 [] 1aw2q(x.1)))     --> (6)
      + 1!_qq? ; R((c/1_1 [] 1dw0q))     --> (7)

(6)  R((c/1_1 [] 1aw2q(x.1))) =
      + 1a_ii!(x.1) ; R((c/2_1(x.1) [] 1a/2(x 1)))     --> (14)
      + #(a.1) . 1a_rr!(x.1,a.1) ; R((c/1_1 [] 1a/1))     --> (5)

(7)  R((c/1_1 [] 1dw0q)) =
      + 1!_ii! ; R((c/0 [] 1a/0))     --> (8)
      + #(!.1)  1!_rr!(!.1) ; R((c/1_1 [] 1a/1))     --> (5)

(8)  R((i5(x.1) [] 1aw2(x.1))) =
      + 1a_ii!(x.1) ; R((c/2_1(x.1) [] 1a/2(x 1)))     --> (14)

(9)  R((r5(x.1) [] 1aw2(x.1))) =
      + #(a.1) . 1a_rr!(x 1,a.1) ; R((c/1_1 [] 1a/1))     --> (5)

(10)  R((i4 [] 1dw0)) =
      + 1!_ii! ; R((c/0 [] 1a/0))     --> (8)

(11)  R((r4 [] 1dw0)) =
      + #(!.1) . 1!_rr!(! 1) ; R((c/1_1 [] 1a/1))     --> (5)

(12)  R((c/1_1 [] 1a/1qqq(x.1,a 1))) =
      + 1a_rr!(x.1,a 1) ; R((c/1_1 [] 1a/1))     --> (5)

(13)  R((c/1_1 [] 1a/1qq(! 1))) =
      + 1!_rr!(!.1) ; R((c/1_1 [] 1a/1))     --> (5)

(14)  R((c/2_1(x.1) [] 1a/2(x 1))) =
      + 1d_qq?(x.1) ; R((c/2_1(x.1) [] 1dw1q(x.1)))     --> (15)

(15)  R((c/2_1(x.1) [] 1dw1q(x.1))) =
      + 1d_ii!(x.1) ; R((c/1_1 [] 1a/1))     --> (5)
      + #(d 1)  1d_rr!(x 1,d.1) ; R((c/2_1(x.1) [] 1a/2(x.1)))     --> (14)
```

(16) R((i13(x.1) [] 1dw1(x.1))) =

+ 1d_ii!(x.1) ; R((c/1_1 [] 1a/1)) --> (5)

(17) R((r13(x.1) [] 1dw1(x.1))) =

+ #(d.1) . 1d_rr'(x.1,d.1) ; R((c/2_1(x.1) [] 1a/2(x 1))) --> (14)

(18) R((c/2_1(x.1) [] 1a/2qq(x.1,d.1))) =

+ 1d_rr'(x.1,d.1) ; R((c/2_1(x.1) [] 1a/2(x.1))) --> (14)

Appendix B. DCP Expressions and Results for Membership-Based System

BRIDGE

(0) c/0 =

+ #(x.1,y.1,n.1) . a_q?(x 1,y.1,n 1) , c/1_ai(x.1,y 1,n.1) --> (1)

+ #(x.1,y.1,n.1) . a_q?(x.1,y.1,n.1) , c/1_ar(x 1,y.1,n 1) --> (2)

(1) c/1_ai(x.1,y.1,n.1) =

+ #(g1.1) . a_i!(x.1,y.1,n.1) , c/1(g1 1) --> (3)

(2) c/1_ar(x.1,y.1,n.1) =

+ #(a.1) . a_r!(x.1,y.1,n.1,a.1) ; c/0 --> (0)

(3) c/1(g1 1) =

+ #(x.1,y.1,n.1) a_q?(x.1,y.1,n 1) ; c/2_ai(x.1,y.1,n.1,g1 1) --> (4)

+ #(x.1,y.1,n.1) a_q?(x.1,y.1,n.1) , c/2_ar(x.1,y.1,n.1,g1 1) --> (5)

+ #(x.1,y 1,n.1) . d_q?(x 1,y.1,n.1) ; c/0_di(x.1,y.1,n.1,g1 1) --> (6)

+ #(x 1,y.1,n.1) d_q?(x.1,y 1,n 1) ; c/0_dr(x 1,y.1,n.1,g1 1) --> (7)

(4) c/2_ai(x.1,y 1,n 1,g1 1) =

+ #(g2.1) . a_i!(x 1,y.1,n.1) , c/2(g2 1) --> (8)

(5) c/2_ar(x 1,y.1,n 1,g1 1) =

+ #(a.1) . a_r!(x.1,y 1,n 1,a 1) , c/1(g1 1) --> (3)

(6) c/0_di(x.1,y.1,n.1,g1.1) =

+ d_i!(x 1,y 1,n.1) , c/0 --> (0)

(7) c/0_dr(x.1,y.1,n 1,g1 1) =

+ #(d.1) d_r!(x 1,y 1,n.1,d 1) ; c/1(g1.1) --> (3)

(8) c/2(g2 1) =

+ #(x.1,y.1,n 1) a_q?(x.1,y 1,n 1) ; c/3_ai(x.1,y.1,n.1,g2.1) --> (9)

+ #(x 1,y.1,n.1) a_q?(x 1,y.1,n 1) , c/3_ar(x.1,y.1,n.1,g2.1) --> (10)

+ #(x 1,y.1,n 1) d_q?(x.1,y 1,n.1) , c/1_di(x.1,y.1,n.1,g2.1) --> (11)

+ #(x 1,y 1,n 1) d_q?(x.1,y 1,n 1) , c/1_dr(x 1,y 1,n.1,g2.1) --> (12)

(9) c/3_ai(x.1,y.1,n.1,g2.1) =

+ #(g3.1) . a_i'(x.1,y.1,n.1) , c/3(g3.1) --> (13)

(10) c/3_ar(x.1,y.1,n.1,g2.1) =

+ #(a.1) . a_r'(x.1,y.1,n.1,a.1) ; c/2(g2.1) --> (8)

(11) c/1_di(x.1,y.1,n.1,g2.1) =

+ #(g1.1) . d_i'(x.1,y.1,n.1) ; c/1(g1.1) --> (3)

(12) c/1_dr(x 1,y 1,n.1,g2.1) =

+ #(d.1) . d_r'(x.1,y.1,n.1,d.1) ; c/2(g2.1) --> (8)

(13) c/3(g3.1) =

+ #(x.1,y.1,n.1) . a_q?(x.1,y.1,n.1) ; c/4_ai(x.1,y 1,n.1,g3 1) --> (14)

+ #(x.1,y.1,n.1) . a_q?(x.1,y.1,n.1) ; c/4_ar(x.1,y 1,n.1,g3 1) --> (15)

+ #(x.1,y.1,n.1) . d_q?(x 1,y 1,n.1) ; c/2_di(x.1,y 1,n 1,g3 1) --> (16)

+ #(x.1,y.1,n.1) . d_q?(x 1,y.1,n.1) ; c/2_dr(x.1,y.1,n 1,g3.1) --> (17)

(14) c/4_ai(x.1,y.1,n.1,g3.1) =

+ #(g4.1) . a_i'(x.1,y.1,n.1) ; c/4(g4.1) --> (18)

(15) c/4_ar(x.1,y 1,n 1,g3.1) =

+ #(a.1) . a_r'(x.1,y 1,n.1,a.1) ; c/3(g3.1) --> (13)

(16) c/2_di(x.1,y.1,n.1,g3 1) =

+ #(g2.1) . d_i'(x.1,y 1,n.1) ; c/2(g2.1) --> (8)

(17) c/2_dr(x.1,y.1,n.1,g3.1) =

+ #(d.1) . d_r'(x.1,y.1,n.1,d 1) , c/3(g3.1) --> (13)

(18) c/4(g4.1) =

+ #(x.1,y.1,n.1) . d_q?(x.1,y.1,n 1) ; c/3_di(x.1,y.1,n.1,g4.1) --> (19)

+ #(x.1,y.1,n.1) . d_q?(x.1,y.1,n.1) ; c/3_dr(x.1,y.1,n.1,g4 1) --> (20)

(19) c/3_di(x.1,y.1,n.1,g4 1) =

+ #(g3.1) . d_r¹(x.1,y 1,n.1) , c/3(g3.1) --> (13)

(20) c/3_dr(x.1,y.1,n.1,g4.1) =

+ #(d.1) . d_r¹(x.1,y.1,n.1,d.1) ; c/4(g4.1) --> (18)

AGENT ----------

(0) a/0(x.1) =

+ #(y.1,n.1) . a_qq²(y.1,n.1) ; aw1q(x.1,y.1,n.1) --> (1)

(1) aw1q(x.1,y.1,n.1) =

+ a_q¹(x.1,y 1,n.1) ; aw1(x.1,y.1,n.1) --> (2)

(2) aw1(x.1,y.1,n.1) =

+ a_i²(x.1,y.1,n.1) ; a/1q(x.1,y.1,n.1) --> (3)

+ #(a.1) . a_r²(x 1,y 1,n.1,a.1) ; a/0q-(x.1,y.1,n.1,a 1) --> (4)

(3) a/1q(x.1,y 1,n 1) =

+ #(m1.1) . a_ii¹(y.1,n 1) , a/1(x 1,m1.1) --> (5)

(4) a/0q-(x.1,y.1,n.1,a 1) =

+ a_rr¹(y.1,n.1,a 1) ; a/0(x 1) --> (0)

(5) a/1(x.1,m1 1) =

+ #(y 1,n.1) . a_qq²(y 1,n 1) ; aw2q(x.1,y.1,n.1,m1.1) --> (6)

+ #(y.1,n 1) . d_qq²(y 1,n 1) , dw0q(x.1,y.1,n.1,m1.1) --> (7)

(6) aw2q(x.1,y.1,n 1,m1 1) =

+ a_q¹(x.1,y 1,n 1) , aw2(x.1,y 1,n 1,m1.1) --> (8)

(7) dw0q(x.1,y.1,n.1,m1.1) =

+ d_q¹(x.1,y.1,n 1) ; dw0(x.1,y 1,n.1,m1.1) --> (9)

(8) aw2(x.1,y 1,n 1,m1 1) =

+ a_i²(x.1,y.1,n.1) , a/2q(x.1,y.1,n.1,m1.1) --> (10)

+ #(a 1) . a_r²(x 1,y 1,n 1,a 1) , a/1q-(x 1,y.1,n.1,a.1,m1.1) --> (11)

(9) dw0(x.1,y.1,n.1,m1.1) =

+ d_i?(x.1,y.1,n.1) ; s/0qq(x.1,y.1,n.1) --> (12)

+ #(d.1) . d_r?(x.1,y.1,n.1,d.1) ; s/1qq-(x.1,y.1,n.1,d.1,m1.1) --> (13)

(10) s/2q(x 1,y.1,n.1,m1.1) =

+ #(m2.1) . s_ii'(y.1,n.1) ; s/2(x.1,m2.1) --> (14)

(11) s/1q-(x.1,y.1,n.1,s.1,m1.1) =

+ s_rr'(y.1,n.1,s.1) ; s/1(x.1,m1.1) --> (5)

(12) s/0qq(x.1,y.1,n.1) =

+ d_ii'(y.1,n.1) ; s/0(x.1) --> (0)

(13) s/1qq-(x.1,y.1,n.1,d.1,m1.1) =

+ d_rr'(y.1,n.1,d.1) ; s/1(x.1,m1.1) --> (5)

(14) s/2(x.1,m2.1) =

+ #(y.1,n.1) . d_qq?(y.1,n.1) ; dw1q(x.1,y.1,n.1,m2.1) --> (15)

(15) dw1q(x.1,y 1,n 1,m2 1) =

+ d_q!(x.1,y.1,n.1) ; dw1(x 1,y 1,n.1,m2.1) --> (16)

(16) dw1(x.1,y.1,n.1,m2.1) =

+ d_i?(x.1,y.1,n.1) ; s/1qq(x.1,y 1,n.1,m2.1) --> (17)

+ #(d.1) . d_r?(x.1,y 1,n.1,d.1) ; s/2qq-(x 1,y.1,n.1,d.1,m2.1) --> (18)

(17) s/1qq(x.1,y 1,n.1,m2.1) =

+ #(m1.1) . d_ii!(y 1,n 1) ; s/1(x.1,m1.1) --> (5)

(18) s/2qq-(x.1,y.1,n 1,d.1,m2 1) =

+ d_rr'(y.1,n.1,d 1) ; s/2(x.1,m2.1) --> (14)

BRIDGE [] AGENT ----------

(0) (c/0 [] s/0(x 1)) =

```
        + #(y.1,n.1) . a_qq'(y.1,n 1) , (c/0 [] aw1q(x.1,y.1,n.1))      --> ( 1)

( 1)  (c/0 [] aw1q(x 1,y.1,n 1)) =
        + a_q(x.1,y.1,n.1) , (c/1_ai(x 1,y.1,n.1) [] aw1(x.1,y.1,n.1))      --> ( 2)
        + a_q(x 1,y.1,n 1) , (c/1_ar(x 1,y.1,n.1) [] aw1(x.1,y.1,n.1))      --> ( 3)

( 2)  (c/1_ai(x.1,y.1,n.1) [] aw1(x 1,y 1,n.1)) =
        + #(g1 1)  a_i(x.1,y.1,n.1) , (c/1(g1.1) [] a/1q(x.1,y.1,n.1))      --> ( 4)

( 3)  (c/1_ar(x 1,y.1,n.1) [] aw1(x.1,y.1,n 1)) =
        + #(a.1) . a_r(x 1,y.1,n.1,a.1) ; (c/0 [] a/0q-(x.1,y.1,n.1,a.1))      --> ( 5)

( 4)  (c/1(g1.1) [] a/1q(x.1,y.1,n.1)) =
        + #(m1 1) . a_ii'(y.1,n.1) ; (c/1(g1.1) [] a/1(x.1,m1 1))      --> ( 6)

( 5)  (c/0 [] a/0q-(x.1,y.1,n 1,a 1)) =
        + a_rr!(y 1,n 1,a.1) ; (c/0 [] a/0(x 1))      --> ( 0)

( 6)  (c/1(g1.1) [] a/1(x 1,m1 1)) =
        + #(y.1,n.1)  a_qq'(y.1,n.1) , (c/1(g1 1) [] aw2q(x.1,y.1,n 1,m1.1))      --> ( 7)
        + #(y.1,n.1)  d_qq'(y 1,n.1) , (c/1(g1 1) [] dw0q(x.1,y.1,n.1,m1.1))      --> ( 8)

( 7)  (c/1(g1 1) [] aw2q(x.1,y.1,n.1,m1.1)) =
        + a_q(x.1,y.1,n.1) ; (c/2_ai(x.1,y 1,n 1,g1 1) [] aw2(x.1,y.1,n.1,m1.1))      --> ( 9)
        + a_q(x.1,y.1,n.1) ; (c/2_ar(x 1,y.1,n 1,g1.1) [] aw2(x 1,y 1,n 1,m1.1))      --> ( 10)

( 8)  (c/1(g1.1) [] dw0q(x.1,y 1,n 1,m1 1)) =
        + d_q(x.1,y 1,n.1) ; (c/0_di(x.1,y 1,n.1,g1 1) [] dw0(x.1,y 1,n.1,m1.1))      --> ( 11)
        + d_q(x 1,y 1,n.1) ; (c/0_dr(x 1,y 1,n 1,g1 1) [] dw0(x.1,y.1,n.1,m1.1))      --> ( 12)

( 9)  (c/2_ai(x.1,y 1,n.1,g1 1) [] aw2(x.1,y 1,n 1,m1 1)) =
        + #(g2.1) . a_i(x 1,y 1,n.1) ; (c/2(g2.1) [] a/2q(x 1,y.1,n.1,m1.1))      --> ( 13)

( 10)  (c/2_ar(x 1,y 1,n 1,g1 1) [] aw2(x 1,y 1,n.1,m1.1)) =
        + #(a 1) . a_r(x 1,y.1,n 1,a 1) ; (c/1(g1 1) [] a/1q-(x.1,y 1,n.1,a.1,m1.1))      --> ( 14)
```

```
( 11)  (c/0_di(x 1,y.1,n 1,g1 1) [] dw0(x 1,y 1,n 1,m1.1)) =
         + d_i(x.1,y 1,n.1) , (c/0 [] a/0qq(x 1,y 1,n 1))        --> ( 15)

( 12)  (c/0_dr(x.1,y.1,n 1,g1 1) [] dw0(x 1,y 1,n 1,m1 1)) =
         + #(d.1)   d_r(x 1,y 1,n.1,d 1) , (c/1(g1 1) [] a/1qq-(x 1,y 1,n.1,d 1,m1 1))        --> ( 16)

( 13)  (c/2(g2.1) [] a/2q(x 1,y 1,n 1,m1 1)) =
         + #(m2.1) . a_ii'(y.1,n 1) , (c/2(g2 1) [] a/2(x 1,m2 1))        --> ( 17)

( 14)  (c/1(g1.1) [] a/1q-(x 1,y 1,n.1,a 1,m1 1)) =
         + a_rr'(y 1,n 1,a.1) , (c/1(g1 1) [] a/1(x 1,m1 1))        --> ( 6)

( 15)  (c/0 [] a/0qq(x 1,y 1,n 1)) =
         + d_ii'(y 1,n 1) , (c/0 [] a/0(x.1))        --> ( 0)

( 16)  (c/1(g1.1) [] a/1qq-(x 1,y 1,n 1,d 1,m1.1)) =
         + d_rr'(y 1,n.1,d 1) , (c/1(g1 1) [] a/1(x 1,m1 1))        --> ( 6)

( 17)  (c/2(g2.1) [] a/2(x 1,m2.1)) =
         + #(y.1,n.1)   d_qq'(y 1,n 1) , (c/2(g2 1) [] dw1q(x 1,y 1,n 1,m2 1))        --> ( 18)

( 18)  (c/2(g2.1) [] dw1q(x 1,y 1,n 1,m2 1)) =
         + d_q(x 1,y 1,n 1) , (c/1_di(x.1,y 1,n 1,g2 1) [] dw1(x.1,y.1,n.1,m2 1))        --> ( 19)
         + d_q(x 1,y 1,n 1) , (c/1_dr(x 1,y 1,n 1,g2 1) [] dw1(x.1,y.1,n 1,m2.1))        --> ( 20)

( 19)  (c/1_di(x 1,y 1,n 1,g2.1) [] dw1(x 1,y 1,n 1,m2 1)) =
         + #(g1 1)   d_i(x 1,y 1,n 1) , (c/1(g1 1) [] a/1qq(x 1,y 1,n 1,m2 1))        --> ( 21)

( 20)  (c/1_dr(x.1,y 1,n.1,g2 1) [] dw1(x 1,y 1,n.1,m2 1)) =
         + #(d 1)   d_r(x 1,y 1,n.1,d 1) , (c/2(g2 1) [] a/2qq-(x 1,y 1,n 1,d 1,m2 1))        --> ( 22)

( 21)  (c/1(g1.1) [] a/1qq(x.1,y 1,n 1,m2 1)) =
         + #(m1.1) . d_ii'(y.1,n.1) ; (c/1(g1 1) [] a/1(x 1,m1.1))        --> ( 6)

( 22)  (c/2(g2.1) [] a/2qq-(x 1,y 1,n 1,d 1,m2 1)) =
```

```
        + d_rr'(y.1,n.1,d 1) ; (c/2(g2 1) [] a/2(x 1,m2 1))      --> ( 17)

R (BRIDGE [] AGENT) ----------------------------------------------------------

( 0)  R((c/0 [] a/0(x.1))) =

        + #(y.1,n.1) . a_qq?(y.1,n.1) , R((c/0 [] aw1q(x 1,y 1,n 1)))      --> ( 1)

( 1)  R((c/0 [] aw1q(x 1,y 1,n.1))) =

        + #(g1.1,m1.1)   a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> ( 6)
        + #(a.1)   a_rr'(y 1,n 1,a 1) , R((c/0 [] a/0(x 1)))      --> ( 0)

( 2)  R((c/1_ai(x 1,y.1,n 1) [] aw1(x 1,y.1,n 1))) =

        + #(g1 1,m1.1)   a_ii'(y.1,n 1) , R((c/1(g1 1) [] a/1(x.1,m1 1)))      --> ( 6)

( 3)  R((c/1_ar(x 1,y 1,n 1) [] aw1(x 1,y 1,n 1))) =

        + #(a 1)   a_rr'(y 1,n 1,a 1) , R((c/0 [] a/0(x.1)))      --> ( 0)

( 4)  R((c/1(g1.1) [] a/1q(x 1,y 1,n 1))) =

        + #(m1 1) . a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/1(x.1,m1 1)))      --> ( 6)

( 5)  R((c/0 [] a/0q-(x 1,y 1,n 1,a 1))) =

        + a_rr'(y 1,n.1,a 1) , R((c/0 [] a/0(x 1)))      --> ( 0)

( 6)  R((c/1(g1 1) [] a/1(x 1,m1 1))) =

        + #(y 1,n.1)   a_qq?(y 1,n 1) , R((c/1(g1 1) [] aw2q(x 1,y 1,n 1,m1 1)))      --> ( 7)
        + #(y.1,n.1)   d_qq?(y 1,n 1) , R((c/1(g1 1) [] dw0q(x.1,y 1,n 1,m1 1)))      --> ( 8)

( 7)  R((c/1(g1 1) [] aw2q(x 1,y 1,n 1,m1 1))) =

        + #(g2.1,m2.1)   a_ii'(y 1,n 1) , R((c/2(g2 1) [] a/2(x 1,m2 1)))      --> ( 17)
        + #(a 1)   a_rr'(y 1,n 1,a 1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> ( 6)

( 8)  R((c/1(g1 1) [] dw0q(x 1,y 1,n 1,m1 1))) =

        + d_ii'(y.1,n 1) ; R((c/0 [] a/0(x 1)))      --> ( 0)
        + #(d.1)   d_rr'(y 1,n.1,d 1) , R((c/1(g1 1) [] a/1(x.1,m1 1)))      --> ( 6)

( 9)  R((c/2_ai(x.1,y 1,n 1,g1 1) [] aw2(x 1,y 1,n 1,m1.1))) =
```

```
        + #(g2.1,m2.1)   a_ii'(y 1,n.1) ; R((c/2(g2 1) [] a/2(x 1,m2 1)))      --> ( 17)

( 10)  R((c/2_ar(x.1,y 1,n 1,g1 1) [] aw2(x 1,y 1,n 1,m1 1))) =
        + #(a.1) . a_rr'(y 1,n 1,a 1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> ( 6)

( 11)  R((c/0_di(x.1,y.1,n 1,g1.1) [] dw0(x.1,y 1,n 1,m1.1))) =
        + d_ii'(y.1,n 1) , R((c/0 [] a/0(x 1)))      --> ( 0)

( 12)  R((c/0_dr(x 1,y 1,n 1,g1.1) [] dw0(x 1,y 1,n 1,m1 1))) =
        + #(d.1)   d_rr'(y.1,n 1,d.1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> ( 6)

( 13)  R((c/2(g2.1) [] a/2q(x.1,y.1,n 1,m1 1))) =
        + #(m2.1)   a_ii'(y.1,n 1) ; R((c/2(g2.1) [] a/2(x 1,m2.1)))      --> ( 17)

( 14)  R((c/1(g1.1) [] a/1q-(x 1,y 1,n.1,a.1,m1 1))) =
        + a_rr'(y.1,n 1,a 1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> ( 6)

( 15)  R((c/0 [] a/0qq(x 1,y.1,n 1))) =
        + d_ii'(y.1,n.1) , R((c/0 [] a/0(x 1)))      --> ( 0)

( 16)  R((c/1(g1 1) [] a/1qq-(x 1,y 1,n 1,d 1,m1.1))) =
        + d_rr'(y.1,n.1,d 1) , R((c/1(g1 1) [] a/1(x 1,m1.1)))      --> ( 6)

( 17)  R((c/2(g2 1) [] a/2(x.1,m2 1))) =
        + #(y 1,n.1)   d_qq?(y 1,n 1) ; R((c/2(g2 1) [] dw1q(x.1,y.1,n.1,m2.1)))      --> ( 18)

( 18)  R((c/2(g2.1) [] dw1q(x 1,y 1,n 1,m2 1))) =
        + #(g1 1,m1 1)   d_ii'(y 1,n 1) , R((c/1(g1 1) [] a/1(x.1,m1 1)))      --> ( 6)
        + #(d.1) . d_rr'(y 1,n 1,d 1) ; R((c/2(g2.1) [] a/2(x 1,m2 1)))      --> ( 17)

( 19)  R((c/1_di(x.1,y.1,n.1,g2 1) [] dw1(x.1,y 1,n.1,m2.1))) =
        + #(g1.1,m1 1)   d_ii'(y.1,n 1) ; R((c/1(g1.1) [] a/1(x.1,m1 1)))      --> ( 6)

( 20)  R((c/1_dr(x.1,y.1,n.1,g2 1) [] dw1(x.1,y 1,n.1,m2.1))) =
        + #(d.1) . d_rr'(y.1,n 1,d 1) , R((c/2(g2 1) [] a/2(x 1,m2 1)))      --> ( 17)
```

```
( 21)  R((c/1(g1 1) [] a/1qq(x.1,y.1,n 1,m2 1))) =

        + #(m1.1) . d_ii'(y 1,n.1) , R((c/1(g1 1) [] a/1(x.1,m1 1)))      --> ( 6)

( 22)  R((c/2(g2.1) [] a/2qq-(x 1,y 1,n 1,d 1,m2 1))) =

        + d_rr'(y 1,n 1,d 1) , R((c/2(g2 1) [] a/2(x 1,m2 1)))      --> ( 17)

Starting  pass 1

Starting  pass 2


EQUIVALENCES ---------------------------------------------------------------

== (R((c/1(g1.1) [] a/1q(x.1,y.1,n 1))) , R((c/1_ai(x 1,y 1,n.1) [] aw1(x 1,y 1,n 1))))
    -->   (4) == (2)
== (R((c/2(g2.1) [] a/2q(x.1,y.1,n 1,m1.1))) , R((c/2_ai(x.1,y 1,n.1,g1.1) [] aw2(x 1,y 1,n 1,m1.1))))
    -->   (18) == (9)
== (R((c/0 [] a/0qq(x.1,y.1,n.1))) , R((c/0_di(x 1,y 1,n 1,g1.1) [] dw0(x 1,y 1,n.1,m1.1))))
    -->   (16) == (11)
== (R((c/1(g1.1) [] a/1qq(x.1,y.1,n.1,m2 1))) , R((c/1_di(x 1,y.1,n 1,g2.1) [] dw1(x 1,y 1,n.1,m2 1))))
    -->   (21) == (19)


R-REDUCED ------------------------------------------------------------------

 (0)  R((c/0 [] a/0(x 1)))  =

       + #(y 1,n 1)   a_qq'(y 1,n 1) , R((c/0 [] aw1q(x 1,y 1,n.1)))      --> (1)

 (1)  R((c/0 [] aw1q(x 1,y.1,n 1)))  =

       + #(g1 1,m1.1) * a_ii'(y 1,n 1) , R((c/1(g1.1) [] a/1(x.1,m1.1)))      --> (5)

       + #(a 1)   a_rr'(y 1,n 1,a 1) , R((c/0 [] a/0(x 1)))      --> (0)

 (2)  R((c/1_ai(x.1,y.1,n 1) [] aw1(x 1,y 1,n 1)))  =

       + #(g1 1,m1 1)   a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/1(x 1,m1 1)))      --> (5)

 (3)  R((c/1_ar(x.1,y 1,n 1) [] aw1(x 1,y 1,n 1)))  =

       + #(a.1)   a_rr'(y 1,n 1,a 1) , R((c/0 [] a/0(x.1)))      --> (0)

 (4)  R((c/0 [] a/0q-(x 1,y.1,n 1,a 1)))  =

       + a_rr'(y.1,n 1,a 1) ; R((c/0 [] a/0(x 1)))      --> (0)

 (5)  R((c/1(g1 1) [] a/1(x 1,m1 1)))  =

       + #(y.1,n 1)   a_qq'(y 1,n 1) , R((c/1(g1.1) [] aw2q(x.1,y 1,n 1,m1 1)))      --> (6)
```

```
      + #(y.1,n.1) . d_qq²(y 1,n 1) ; R((c/1(g1.1) [] dw0q(x.1,y 1,n.1,m1 1)))     --> (7)

(6)  R((c/1(g1.1) [] sw2q(x 1,y 1,n 1,m1 1))) =
      + #(g2.1,m2.1)   s_ii'(y 1,n 1) , R((c/2(g2.1) [] s/2(x.1,m2.1)))     --> (14)
      + #(s.1) . s_rr'(y 1,n 1,s.1) ; R((c/1(g1.1) [] s/1(x 1,m1 1)))     --> (5)

(7)  R((c/1(g1.1) [] dw0q(x 1,y.1,n.1,m1.1))) =
      + d_ii'(y.1,n.1) ; R((c/0 [] s/0(x.1)))     --> (0)
      + #(d.1) . d_rr'(y.1,n.1,d.1) ; R((c/1(g1.1) [] s/1(x.1,m1 1)))     --> (5)

(8)  R((c/2_si(x.1,y.1,n 1,g1.1) [] sw2(x.1,y.1,n 1,m1 1))) =
      + #(g2.1,m2.1)   s_ii'(y 1,n 1) ; R((c/2(g2.1) [] s/2(x.1,m2 1)))     --> (14)

(9)  R((c/2_sr(x.1,y 1,n.1,g1.1) [] sw2(x.1,y 1,n 1,m1 1))) =
      + #(s.1) . s_rr'(y.1,n 1,s.1) , R((c/1(g1 1) [] s/1(x.1,m1.1)))     --> (5)

(10)  R((c/0_di(x.1,y 1,n.1,g1 1) [] dw0(x 1,y 1,n 1,m1 1))) =
      + d_ii'(y.1,n.1) , R((c/0 [] s/0(x 1)))     --> (0)

(11)  R((c/0_dr(x.1,y.1,n.1,g1.1) [] dw0(x 1,y.1,n.1,m1.1))) =
      + #(d.1)   d_rr'(y.1,n.1,d.1) ; R((c/1(g1 1) [] s/1(x.1,m1.1)))     --> (5)

(12)  R((c/1(g1 1) [] s/1q-(x.1,y.1,n.1,s 1,m1 1))) =
      + s_rr'(y.1,n 1,s 1) ; R((c/1(g1 1) [] s/1(x.1,m1 1)))     --> (5)

(13)  R((c/1(g1.1) [] s/1qq-(x.1,y.1,n 1,d.1,m1 1))) =
      + d_rr'(y.1,n.1,d 1) ; R((c/1(g1.1) [] s/1(x 1,m1.1)))     --> (5)

(14)  R((c/2(g2.1) [] s/2(x 1,m2.1))) =
      + #(y.1,n.1) . d_qq²(y.1,n.1) ; R((c/2(g2.1) [] dw1q(x.1,y.1,n.1,m2.1)))     --> (15)

(15)  R((c/2(g2.1) [] dw1q(x.1,y.1,n 1,m2.1))) =
      + #(g1.1,m1.1)   d_ii'(y.1,n.1) , R((c/1(g1 1) [] s/1(x.1,m1.1)))     --> (5)
      + #(d.1)   d_rr'(y.1,n 1,d.1) , R((c/2(g2 1) [] s/2(x 1,m2.1)))     --> (14)

(16)  R((c/1_di(x.1,y.1,n.1,g2 1) [] dw1(x 1,y.1,n 1,m2.1))) =
```

+ #(g1.1,m1.1) . d_ii'(y 1,n.1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(17) R((c/1_dr(x.1,y.1,n.1,g2.1) [] dw1(x 1,y.1,n.1,m2.1))) =

+ #(d.1) . d_rr'(y.1,n 1,d.1) ; R((c/2(g2.1) [] a/2(x.1,m2 1))) --> (14)

(18) R((c/2(g2.1) [] a/2qq-(x.1,y.1,n.1,d 1,m2.1))) =

+ d_rr'(y.1,n.1,d 1) , R((c/2(g2.1) [] a/2(x.1,m2.1))) --> (14)

Appendix C. DCP Expressions and Results for Conference-Based System

BRIDGE

(0) c/0 =

+ #(x.1,y.1,n.1) . a_q?(x 1,y.1,n 1) , c/1_ai(x.1,y.1,n 1) --> (1)

+ #(x.1,y.1,n 1) a_q?(x 1,y.1,n 1) ; c/1_ar(x 1,y 1,n.1) --> (2)

(1) c/1_ai(x.1,y.1,n.1) =

+ #(g1.1,t3.1) . a_i!(x.1,y.1,n 1,t3 1) , c/1(g1 1) --> (3)

(2) c/1_ar(x.1,y.1,n.1) =

+ #(a.1) . a_r!(x.1,y 1,n 1,a.1) ; c/0 --> (0)

(3) c/1(g1.1) =

+ #(x.1,y.1,n.1) a_q?(x 1,y 1,n 1) ; c/2_ai(x.1,y.1,n.1,g1.1) --> (4)

+ #(x 1,y.1,n 1) a_q?(x.1,y 1,n.1) , c/2_ar(x.1,y.1,n 1,g1 1) --> (5)

+ #(x 1,y 1,n 1) d_q?(x.1,y.1,n 1) ; c/0_di(x 1,y.1,n.1,g1.1) --> (6)

+ #(x.1,y.1,n.1) d_q?(x.1,y 1,n.1) ; c/0_dr(x.1,y.1,n.1,g1.1) --> (7)

(4) c/2_ai(x 1,y.1,n.1,g1 1) =

+ #(g2.1,t3 1) . a_i!(x 1,y 1,n 1,t3 1) , c/2(g2 1) --> (8)

+ #(t4.1) a_i!(x 1,y.1,n 1,t4 1) ; c/1(g1 1) --> (3)

(5) c/2_ar(x 1,y 1,n 1,g1 1) =

+ #(a.1) a_r!(x 1,y 1,n.1,a 1) ; c/1(g1 1) --> (3)

(6) c/0_di(x.1,y.1,n 1,g1.1) =

+ #(t1.1) . d_i!(x.1,y 1,n 1,t1 1) , c/0 --> (0)

+ #(t2.1) . d_i!(x 1,y.1,n.1,t2.1) ; c/1(g1.1) --> (3)

(7) c/0_dr(x.1,y.1,n.1,g1.1) =

+ #(d.1) . d_r!(x 1,y.1,n 1,d 1) ; c/1(g1.1) --> (3)

(8) c/2(g2.1) =

+ #(x 1,y.1,n 1) a_q?(x 1,y 1,n 1) , c/3_ai(x.1,y.1,n 1,g2 1) --> (9)

+ #(x.1,y 1,n 1) a_q?(x 1,y 1,n 1) ; c/3_ar(x.1,y.1,n.1,g2.1) --> (10)

```
      + #(x.1,y.1,n.1) . d_q?(x.1,y.1,n 1) ; c/1_di(x.1,y 1,n 1,g2.1)     --> (11)
      + #(x.1,y.1,n.1) . d_q?(x.1,y 1,n.1) ; c/1_dr(x.1,y 1,n.1,g2 1)     --> (12)

(9)  c/3_si(x.1,y.1,n 1,g2.1)  =
      + #(t4.1) . s_i!(x.1,y.1,n.1,t4.1) ; c/2(g2 1)     --> (8)

(10)  c/3_sr(x.1,y.1,n.1,g2.1)  =
      + #(s.1) . s_r!(x.1,y.1,n.1,s 1) , c/2(g2 1)     --> (8)

(11)  c/1_di(x.1,y 1,n.1,g2.1)  =
      + #(g1.1,t1.1) . d_i!(x 1,y.1,n.1,t1.1) , c/1(g1 1)     --> (3)
      + #(t2.1)  d_i!(x.1,y 1,n.1,t2.1) , c/2(g2.1)     --> (8)

(12)  c/1_dr(x.1,y 1,n 1,g2.1)  =
      + #(d.1) . d_r!(x 1,y.1,n 1,d 1) , c/2(g2.1)     --> (8)

AGENT ------------------------------------------------------------------------

(0)  s/0(x 1)  =
      + #(y.1,n.1)  s_qq?(y.1,n.1) , sw1q(x.1,y.1,n.1)     --> (1)

(1)  sw1q(x 1,y.1,n 1)  =
      + s_q!(x.1,y.1,n.1) ; sw1(x 1,y 1,n.1)     --> (2)

(2)  sw1(x.1,y.1,n.1)  =
      + #(t3.1) . s_i?(x 1,y 1,n.1,t3 1) ; s/1q(x.1,y 1,n.1)     --> (3)
      + #(s.1)  s_r?(x 1,y.1,n 1,s.1) ; s/0q-(x.1,y 1,n.1,s 1)     --> (4)

(3)  s/1q(x.1,y.1,n.1)  =
      + #(m1.1) . s_ii!(y.1,n.1) , s/1(x.1,m1.1)     --> (5)

(4)  s/0q-(x.1,y 1,n.1,s 1)  =
      + s_rr!(y.1,n.1,s.1) ; s/0(x.1)     --> (0)

(5)  s/1(x.1,m1.1)  =
      + #(y 1,n 1) . s_qq?(y.1,n.1) ; sw2q(x.1,y 1,n 1,m1.1)     --> (6)
```

+ #(y 1,n 1) d_qq²(y.1,n 1) , dw0q(x.1,y.1,n 1,m1.1) --> (7)

(6) sw2q(x.1,y.1,n.1,m1.1) =

+ a_q¹(x.1,y.1,n.1) ; sw2(x.1,y.1,n 1,m1 1) --> (8)

(7) dw0q(x.1,y.1,n.1,m1.1) =

+ d_q¹(x.1,y.1,n.1) ; dw0(x.1,y.1,n.1,m1.1) --> (9)

(8) sw2(x 1,y.1,n.1,m1 1) =

+ #(t4.1) . a_i²(x.1,y.1,n.1,t4.1) ; s/2q(x.1,y.1,n 1,m1 1) --> (10)

+ #(s.1) . a_r²(x.1,y 1,n.1,s.1) ; s/1q-(x.1,y 1,n.1,s 1,m1 1) --> (11)

(9) dw0(x.1,y.1,n.1,m1.1) =

+ #(t1.1) . d_i²(x.1,y.1,n.1,t1 1) ; s/0qq(x.1,y 1,n 1) --> (12)

+ #(d.1) . d_r²(x.1,y.1,n.1,d 1) , s/1qq-(x 1,y 1,n.1,d 1,m1 1) --> (13)

(10) s/2q(x.1,y.1,n.1,m1.1) =

+ #(m2.1) a_ii¹(y 1,n.1) ; s/2(x.1,m2.1) --> (14)

(11) s/1q-(x 1,y.1,n 1,s.1,m1.1) =

+ a_rr¹(y.1,n.1,s 1) , s/1(x.1,m1.1) --> (5)

(12) s/0qq(x 1,y 1,n 1) =

+ d_ii¹(y 1,n 1) , s/0(x 1) --> (0)

(13) s/1qq-(x 1,y 1,n 1,d.1,m1.1) =

+ d_rr¹(y 1,n 1,d 1) ; s/1(x 1,m1 1) --> (5)

(14) s/2(x.1,m2.1) =

+ #(y.1,n.1) . d_qq²(y.1,n 1) ; dw1q(x.1,y.1,n 1,m2 1) --> (15)

(15) dw1q(x.1,y.1,n.1,m2.1) =

+ d_q¹(x 1,y 1,n.1) ; dw1(x.1,y 1,n 1,m2 1) --> (16)

(16) dw1(x 1,y 1,n 1,m2 1) =

```
      + #(t2.1) . d_i?(x.1,y.1,n.1,t2.1) ; s/1qq(x.1,y 1,n.1,m2 1)      --> (17)
      + #(t1.1) . d_i?(x.1,y.1,n.1,t1.1) ; s/0qq(x 1,y 1,n 1)      --> (12)
      + #(d.1) . d_r?(x.1,y.1,n.1,d 1) ; s/2qq-(x.1,y.1,n.1,d.1,m2.1)      --> (18)

(17)  s/1qq(x.1,y.1,n.1,m2.1)  =
      + #(m1.1) . d_ii!(y.1,n.1) ; s/1(x.1,m1.1)      --> (5)

(18)  s/2qq-(x.1,y.1,n.1,d.1,m2.1)  =
      + d_rr!(y.1,n.1,d 1) ; s/2(x.1,m2.1)      --> (14)

BRIDGE [] AGENT ----------------------------------------------------------------

( 0)  (c/0 [] s/0(x.1)) =
      + #(y.1,n 1) . s_qq?(y.1,n.1) ; (c/0 [] sw1q(x 1,y 1,n.1))      --> ( 1)

( 1)  (c/0 [] sw1q(x.1,y.1,n.1)) =
      + s_q(x.1,y 1,n.1) , (c/1_si(x.1,y.1,n.1) [] sw1(x.1,y.1,n 1))      --> ( 2)
      + s_q(x.1,y 1,n.1) ; (c/1_sr(x.1,y.1,n.1) [] sw1(x.1,y.1,n.1))      --> ( 3)

( 2)  (c/1_si(x.1,y.1,n.1) [] sw1(x 1,y.1,n.1)) =
      + #(t3.1,g1 1) . s_i(x 1,y 1,n.1,t3 1) , (c/1(g1.1) [] s/1q(x.1,y.1,n.1))      --> ( 4)

( 3)  (c/1_sr(x.1,y.1,n 1) [] sw1(x 1,y.1,n 1)) =
      + #(s 1) . s_r(x.1,y 1,n.1,s 1) ; (c/0 [] s/0q-(x.1,y.1,n.1,s 1))      --> ( 5)

( 4)  (c/1(g1.1) [] s/1q(x.1,y 1,n 1)) =
      + #(m1.1) . s_ii!(y 1,n.1) ; (c/1(g1.1) [] s/1(x.1,m1.1))      --> ( 6)

( 5)  (c/0 [] s/0q-(x.1,y.1,n 1,s.1)) =
      + s_rr!(y.1,n.1,s.1) , (c/0 [] s/0(x.1))      --> ( 0)

( 6)  (c/1(g1.1) [] s/1(x.1,m1.1)) =
      + #(y.1,n.1) . s_qq?(y.1,n.1) ; (c/1(g1 1) [] sw2q(x.1,y.1,n.1,m1.1))      --> ( 7)
      + #(y.1,n.1) . d_qq?(y.1,n.1) ; (c/1(g1.1) [] dw0q(x.1,y.1,n.1,m1.1))      --> ( 8)

( 7)  (c/1(g1.1) [] sw2q(x.1,y.1,n.1,m1.1)) =
```

+ a_q(x.1,y.1,n.1) ; (c/2_ai(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n.1,m1.1)) --> (9)

+ a_q(x.1,y.1,n.1) ; (c/2_ar(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n.1,m1.1)) --> (10)

(8) (c/1(g1.1) [] dw0q(x.1,y.1,n.1,m1.1)) =

+ d_q(x.1,y.1,n.1) ; (c/0_di(x.1,y.1,n.1,g1.1) [] dw0(x.1,y.1,n.1,m1.1)) --> (11)

+ d_q(x.1,y.1,n.1) ; (c/0_dr(x.1,y.1,n.1,g1.1) [] dw0(x.1,y.1,n.1,m1.1)) --> (12)

(9) (c/2_ai(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n.1,m1.1)) =

+ #(t4.1) . a_i(x.1,y.1,n.1,t4.1) ; (c/1(g1.1) [] a/2q(x.1,y.1,n.1,m1.1)) --> (13).

(10) (c/2_ar(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n.1,m1.1)) =

+ #(a.1) a_r(x.1,y.1,n.1,a.1) ; (c/1(g1.1) [] a/1q-(x.1,y.1,n.1,a.1,m1.1)) --> (14)

(11) (c/0_di(x.1,y.1,n.1,g1.1) [] dw0(x.1,y.1,n.1,m1.1)) =

+ #(t1.1) . d_i(x.1,y.1,n.1,t1.1) ; (c/0 [] a/0qq(x.1,y.1,n.1)) --> (15)

(12) (c/0_dr(x.1,y.1,n.1,g1.1) [] dw0(x.1,y.1,n.1,m1.1)) =

+ #(d.1) . d_r(x.1,y.1,n.1,d.1) ; (c/1(g1.1) [] a/1qq-(x.1,y.1,n.1,d.1,m1.1)) --> (16)

(13) (c/1(g1.1) [] a/2q(x.1,y.1,n.1,m1.1)) =

+ #(m2.1) . a_ii'(y.1,n.1) ; (c/1(g1.1) [] a/2(x.1,m2.1)) --> (17)

(14) (c/1(g1.1) [] a/1q-(x.1,y.1,n.1,a.1,m1.1)) =

+ a_rr'(y.1,n.1,a.1) , (c/1(g1.1) [] a/1(x.1,m1.1)) --> (8)

(15) (c/0 [] a/0qq(x.1,y.1,n.1)) =

+ d_ii'(y.1,n.1) ; (c/0 [] a/0(x.1)) --> (0)

(16) (c/1(g1.1) [] a/1qq-(x.1,y.1,n.1,d.1,m1.1)) =

+ d_rr'(y.1,n.1,d.1) ; (c/1(g1.1) [] a/1(x.1,m1.1)) --> (8)

(17) (c/1(g1.1) [] a/2(x.1,m2.1)) =

+ #(y.1,n.1) . d_qq'(y.1,n.1) ; (c/1(g1.1) [] dw1q(x.1,y.1,n.1,m2.1)) --> (18)

(18) (c/1(g1.1) [] dw1q(x.1,y.1,n.1,m2.1)) =

+ d_q(x.1,y.1,n.1) ; (c/0_di(x.1,y.1,n.1,g1.1) [] dw1(x.1,y.1,n.1,m2.1)) --> (19).

```
        + d_q(x.1,y.1,n.1) ; (c/0_dr(x 1,y.1,n.1,g1.1) [] dw1(x.1,y.1,n.1,m2.1))      --> ( 20)

( 19)  (c/0_di(x.1,y.1,n.1,g1.1) [] dw1(x 1,y.1,n 1,m2 1)) =
        + #(t1.1) . d_i(x.1,y.1,n.1,t1.1) ; (c/0 [] a/0qq(x.1,y.1,n 1))      --> ( 15)
        + #(t2 1) . d_i(x.1,y.1,n.1,t2.1) ; (c/1(g1.1) [] a/1qq(x 1,y.1,n 1,m2.1))      --> ( 21)

( 20)  (c/0_dr(x.1,y.1,n.1,g1 1) [] dw1(x.1,y.1,n 1,m2.1)) =
        + #(d.1)   d_r(x.1,y.1,n.1,d.1) ; (c/1(g1.1) [] a/2qq-(x.1,y.1,n.1,d.1,m2.1))      --> ( 22)

( 21)  (c/1(g1.1) [] a/1qq(x.1,y.1,n 1,m2.1)) =
        + #(m1.1) . d_ii'(y.1,n.1) ; (c/1(g1 1) [] a/1(x 1,m1.1))      --> ( 6)

( 22)  (c/1(g1.1) [] a/2qq-(x.1,y.1,n.1,d 1,m2.1)) =
        + d_rr'(y.1,n.1,d.1) ; (c/1(g1.1) [] a/2(x.1,m2.1))      --> ( 17)

R (BRIDGE [] AGENT) ----------------------------------------------------------

( 0)  R((c/0 [] a/0(x.1))) =
        + #(y.1,n.1) . a_qq?(y 1,n 1) , R((c/0 [] aw1q(x 1,y.1,n.1)))      --> ( 1)

( 1)  R((c/0 [] aw1q(x.1,y.1,n.1))) =
        + #(g1.1,m1 1)   a_ii'(y.1,n.1) ; R((c/1(g1.1) [] a/1(x.1,m1.1)))      --> ( 6)
        + #(a.1) . a_rr'(y 1,n.1,a.1) ; R((c/0 [] a/0(x.1)))      --> ( 0)

( 2)  R((c/1_ai(x.1,y.1,n.1) [] aw1(x.1,y.1,n 1))) =
        + #(g1.1,m1 1) . a_ii'(y.1,n 1) , R((c/1(g1.1) [] a/1(x.1,m1.1)))      --> ( 6)

( 3)  R((c/1_ar(x.1,y.1,n 1) [] aw1(x.1,y.1,n 1))) =
        + #(a.1) . a_rr'(y.1,n.1,a.1) , R((c/0 [] a/0(x 1)))      --> ( 0)

( 4)  R((c/1(g1.1) [] a/1q(x 1,y 1,n.1))) =
        + #(m1.1) . a_ii'(y 1,n.1) ; R((c/1(g1.1) [] a/1(x.1,m1.1)))      --> ( 6)

( 5)  R((c/0 [] a/0q-(x.1,y.1,n.1,a.1))) =
        + a_rr'(y.1,n.1,a.1) ; R((c/0 [] a/0(x.1)))      --> ( 0)
```

(6) R((c/1(g1.1) [] a/1(x.1,m1.1))) =

+ #(y.1,n.1) . a_qq?(y.1,n.1) ; R((c/1(g1.1) [] aw2q(x.1,y.1,n.1,m1.1))) --> (7)

+ #(y 1,n.1) . d_qq?(y.1,n.1) ; R((c/1(g1 1) [] dw0q(x.1,y.1,n.1,m1.1))) --> (8)

(7) R((c/1(g1.1) [] aw2q(x.1,y.1,n 1,m1.1))) =

+ #(m2.1) . a_ii'(y 1,n.1) ; R((c/1(g1.1) [] a/2(x.1,m2.1))) --> (17)

+ #(a.1) . a_rr'(y.1,n.1,a 1) ; R((c/1(g1.1) [] a/1(x.1,m1.1))) --> (6)

(8) R((c/1(g1.1) [] dw0q(x.1,y.1,n.1,m1.1))) =

+ d_ii'(y.1,n.1) ; R((c/0 [] a/0(x.1))) --> (0)

+ #(d.1) . d_rr'(y 1,n.1,d.1) , R((c/1(g1 1) [] a/1(x.1,m1.1))) --> (6)

(9) R((c/2_a(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n 1,m1 1))) =

+ #(m2.1) . a_ii'(y 1,n.1) , R((c/1(g1 1) [] a/2(x.1,m2.1))) --> (17)

(10) R((c/2_ar(x 1,y 1,n 1,g1 1) [] aw2(x 1,y 1,n 1,m1 1))) =

+ #(a.1) a_rr'(y 1,n 1,a.1) ; R((c/1(g1.1) [] a/1(x.1,m1.1))) --> (6)

(11) R((c/0_d(x.1,y 1,n.1,g1 1) [] dw0(x 1,y 1,n 1,m1 1))) =

+ d_ii'(y 1,n.1) ; R((c/0 [] a/0(x 1))) --> (0)

(12) R((c/0_dr(x 1,y.1,n.1,g1.1) [] dw0(x 1,y.1,n 1,m1 1))) =

+ #(d 1) . d_rr'(y 1,n.1,d 1) , R((c/1(g1 1) [] a/1(x.1,m1 1))) --> (6)

(13) R((c/1(g1 1) [] a/2q(x 1,y 1,n 1,m1.1))) =

+ #(m2 1) . a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/2(x.1,m2.1))) --> (17)

(14) R((c/1(g1.1) [] a/1q-(x.1,y 1,n 1,a 1,m1 1))) =

+ a_rr'(y.1,n.1,a.1) , R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (6)

(15) R((c/0 [] a/0qq(x.1,y 1,n 1))) =

+ d_ii'(y.1,n.1) ; R((c/0 [] a/0(x.1))) --> (0)

(16) R((c/1(g1.1) [] a/1qq-(x 1,y.1,n 1,d 1,m1.1))) =

+ d_rr'(y.1,n.1,d 1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (6)

```
( 17)  R((c/1(g1.1) [] a/2(x.1,m2 1))) =
         + #(y.1,n.1)   d_qq'(y.1,n 1) , R((c/1(g1 1) [] dw1q(x 1,y 1,n 1,m2 1)))      --> ( 18)

( 18)  R((c/1(g1.1) [] dw1q(x 1,y.1,n.1,m2 1))) =
         + d_ii'(y.1,n.1) ; R((c/0 [] a/0(x 1)))       --> ( 0)
         + #(m1.1)   d_ii'(y.1,n.1) ; R((c/1(g1 1) [] a/1(x.1,m1 1)))       --> ( 6)
         + #(d.1) . d_rr'(y.1,n.1,d.1) ; R((c/1(g1.1) [] a/2(x 1,m2.1)))       --> ( 17)

( 19)  R((c/0_di(x 1,y.1,n.1,g1.1) [] dw1(x 1,y 1,n 1,m2 1))) =
         + d_ii'(y.1,n.1) ; R((c/0 [] a/0(x 1)))       --> ( 0)
         + #(m1 1) . d_ii'(y.1,n 1) ; R((c/1(g1.1) [] a/1(x 1,m1 1)))       --> ( 6)

( 20)  R((c/0_dr(x.1,y.1,n.1,g1.1) [] dw1(x.1,y.1,n.1,m2.1))) =
         + #(d.1)   d_rr'(y 1,n 1,d 1) , R((c/1(g1.1) [] a/2(x 1,m2.1)))       --> ( 17)

( 21)  R((c/1(g1.1) [] a/1qq(x 1,y.1,n 1,m2 1))) =
         + #(m1.1) . d_ii'(y 1,n.1) ; R((c/1(g1 1) [] a/1(x 1,m1.1)))       --> ( 6)

( 22)  R((c/1(g1.1) [] a/2qq-(x.1,y.1,n.1,d 1,m2 1))) =
         + d_rr'(y.1,n.1,d.1) , R((c/1(g1.1) [] a/2(x 1,m2 1)))       --> ( 17)

Starting  pass 1

Starting  pass 2

Starting  pass 3

Starting  pass 4

EQUIVALENCES ---------------------------------------------------------------

== (R((c/1(g1.1) [] a/1q(x.1,y.1,n.1))) , R((c/1_ai(x 1,y 1,n 1) [] aw1(x 1,y 1,n.1))))
     -->    (4) == (2)
== (R((c/1(g1.1) [] a/2q(x.1,y.1,n.1,m1.1))) , R((c/2_ai(x.1,y.1,n.1,g1.1) [] aw2(x.1,y.1,n.1,m1 1))))
     -->    (13) == (9)
== (R((c/0 [] a/0qq(x.1,y 1,n.1))) , R((c/0_di(x 1,y 1,n.1,g1 1) [] dw0(x.1,y 1,n.1,m1.1))))
     -->    (15) == (11)

R-REDUCED ------------------------------------------------------------------

 (0)  R((c/0 [] a/0(x.1)))  =
```

+ #(y.1,n.1) . a_qq?(y 1,n 1) , R((c/0 [] aw1q(x 1,y 1,n 1))) --> (1)

(1) R((c/0 [] aw1q(x.1,y 1,n 1))) =

+ #(g1.1,m1.1) . a_ii'(y 1,n 1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

+ #(a 1) . a_rr'(y.1,n.1,a.1) , R((c/0 [] a/0(x.1))) --> (0)

(2) R((c/1_ai(x.1,y.1,n.1) [] aw1(x.1,y 1,n 1))) =

+ #(g1.1,m1.1) . a_ii'(y 1,n.1) , R((c/1(g1 1) [] a/1(x 1,m1.1))) --> (5)

(3) R((c/1_ar(x 1,y 1,n 1) [] aw1(x.1,y.1,n 1))) =

+ #(a.1) . a_rr'(y.1,n 1,a.1) ; R((c/0 [] a/0(x 1))) --> (0)

(4) R((c/0 [] a/0q-(x 1,y 1,n 1,a.1))) =

+ a_rr'(y.1,n.1,a 1) ; R((c/0 [] a/0(x 1))) --> (0)

(5) R((c/1(g1.1) [] a/1(x.1,m1 1))) =

+ #(y.1,n.1) . a_qq?(y 1,n.1) , R((c/1(g1 1) [] aw2q(x 1,y 1,n.1,m1 1))) --> (6)

+ #(y.1,n.1) . d_qq?(y 1,n 1) , R((c/1(g1 1) [] dw0q(x 1,y 1,n 1,m1 1))) --> (7)

(6) R((c/1(g1 1) [] aw2q(x 1,y.1,n 1,m1 1))) =

+ #(m2 1) a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/2(x 1,m2.1))) --> (14)

+ #(a.1) . a_rr'(y.1,n 1,a 1) , R((c/1(g1 1) [] a/1(x 1,m1.1))) --> (5)

(7) R((c/1(g1 1) [] dw0q(x 1,y 1,n 1,m1 1))) =

+ d_ii'(y.1,n.1) , R((c/0 [] a/0(x.1))) --> (0)

+ #(d.1) . d_rr'(y 1,n.1,d 1) , R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(8) R((c/2_ai(x 1,y.1,n 1,g1 1) [] aw2(x 1,y 1,n 1,m1 1))) =

+ #(m2 1) a_ii'(y 1,n 1) , R((c/1(g1 1) [] a/2(x 1,m2 1))) --> (14)

(9) R((c/2_ar(x 1,y 1,n 1,g1 1) [] aw2(x 1,y 1,n 1,m1 1))) =

+ #(a 1) a_rr'(y 1,n 1,a 1) , R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(10) R((c/0_di(x 1,y 1,n 1,g1 1) [] dw0(x 1,y 1,n 1,m1 1))) =

+ d_ii'(y 1,n 1) , R((c/0 [] a/0(x 1))) --> (0)

(11) R((c/0_dr(x.1,y.1,n 1,g1.1) [] dw0(x 1,y 1,n 1,m1 1))) =

+ #(d.1) . d_rr!(y.1,n.1,d 1) ; R((c/1(g1.1) [] a/1(x 1,m1.1))) --> (5)

(12) R((c/1(g1.1) [] a/1q-(x.1,y.1,n 1,a 1,m1.1))) =

+ a_rr!(y.1,n.1,a.1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(13) R((c/1(g1.1) [] a/1qq-(x.1,y.1,n.1,d 1,m1 1))) =

+ d_rr!(y.1,n.1,d.1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(14) R((c/1(g1 1) [] a/2(x.1,m2 1))) =

+ #(y.1,n.1) . d_qq?(y.1,n.1) ; R((c/1(g1.1) [] dw1q(x.1,y.1,n.1,m2 1))) --> (15)

(15) R((c/1(g1.1) [] dw1q(x.1,y.1,n.1,m2 1))) =

+ d_ii!(y 1,n.1) , R((c/0 [] a/0(x 1))) --> (0)

+ #(m1.1) . d_ii!(y 1,n.1) , R((c/1(g1 1) [] a/1(x.1,m1.1))) --> (5)

+ #(d.1) . d_rr!(y.1,n 1,d 1) ; R((c/1(g1.1) [] a/2(x.1,m2.1))) --> (14)

(16) R((c/0_di(x.1,y.1,n.1,g1.1) [] dw1(x 1,y 1,n.1,m2.1))) =

+ d_ii!(y.1,n.1) , R((c/0 [] a/0(x 1))) --> (0)

+ #(m1.1) . d_ii!(y.1,n.1) ; R((c/1(g1 1) [] a/1(x 1,m1 1))) --> (5)

(17) R((c/0_dr(x.1,y.1,n.1,g1.1) [] dw1(x.1,y 1,n 1,m2 1))) =

+ #(d.1) . d_rr!(y.1,n 1,d 1) , R((c/1(g1 1) [] a/2(x 1,m2.1))) --> (14)

(18) R((c/1(g1.1) [] a/1qq(x.1,y.1,n 1,m2 1))) =

+ #(m1.1) . d_ii!(y 1,n.1) , R((c/1(g1 1) [] a/1(x.1,m1.1))) --> (5)

(19) R((c/1(g1.1) [] a/2qq-(x.1,y.1,n.1,d.1,m2.1))) =

+ d_rr!(y.1,n.1,d 1) ; R((c/1(g1.1) [] a/2(x.1,m2.1))) --> (14)