Journal of Computer and System Sciences **62**, 73–122 (2001)
doi:10.1006/jcss.2000.1721, available online at http://www.idealibrary.com on IDEAL®

A Process-Calculus Analysis of Concurrent Operations on B-Trees

Anna Philippou

Department of Computer Science, University of Cyprus, Cyprus

and

David Walker

Computing Laboratory, Oxford University, Oxford, United Kingdom

Received April 21, 1998; revised April 14, 2000;
published online December 29, 2000

A general theory of interactive systems with changing structure, the π-calculus, is used to study concurrent operations on a variant of the *B*-tree. An improved algorithm for compression of the data structure is given. Some general results on partially confluent processes and on client-server systems, couched in the π-calculus, are presented.

Key Words: concurrency; data structures; mobility; π-calculus; partial confluence.

1. INTRODUCTION

This paper has two primary goals. The first is to illustrate how the π-calculus, a general theory of interactive systems with changing structure, can be used to reason rigorously about concurrent operations on data structures: a study of operations on B^{link}-trees is presented, and an improved algorithm for compression of the data structure is explained. The second aim is to show some general results on client–server systems, couched in the general theory.

The *B*-tree [1] and variants of it such as the B^*-tree [20] are widely used as index structures. Of concern here is the B^{link}-tree of Lehman and Yao [6] and the algorithms for searching, inserting into, deleting from, and compressing it of [6] and of Sagiv [16]; see also [2, 3, 5, 9] for algorithms on related structures. The data structure and the algorithms are described in Section 3.

The π-calculus [13] allows direct expression of *mobile* systems in whose evolution components can be created and the connections among them change. The primitive entities of the calculus are *names*. Components use names to interact with one another, and by passing names in interactions, components can pass to one

0022-0000/01 $35.00

another the ability to interact with other components. Terms of the calculus are interpreted as labelled transition systems whose points describe system states and whose arrows carry information pertinent to state change. An equivalence relation on transition systems induces a relation capturing when terms express systems having the same observable behaviour.

The B^{link}-system is mobile: when operations are requested, processes are created that are naturally thought of as acting concurrently on a graph, creating nodes, and altering the pointer structure among nodes as they carry out their tasks. In asserting this we are beginning to outline the model we will use, which is quite different from those of [6, 16]. There, operations are expressed using a pseudo-code that contains commands for copying data to and from an implicit secondary storage device (disk) and for locking pages of the disk. Here we express both operations and the data they operate on as active processes, a central idea being that a pointer is represented by a π-calculus name. Further, we model the algorithms more abstractly and thus separate the question of their correctness from that of their correct implementation. In our view the result is descriptions that are clearer and more readily comprehended, once the calculus is familiar. Further, the model is amenable to a perspicuous rigorous analysis that gives insight into why the algorithms are correct and from which we obtain general results.

We use the π-calculus's general notion of behavioural equivalence to express the correctness of the algorithms. We give a very simple term that expresses the expected interactions of the B^{link}-system with its environment (the receipt of requests to carry out operations and the return of the results of doing so). The assertion of correctness is that this term is behaviourally equivalent to the term describing the system. Thus the assertion of correctness is in terms of the observable behaviour of the system, rather than a statement of serializability of computations, of which many varieties have been proposed and used as criteria by which to judge related algorithms. For instance, in [16] a criterion of correctness for the algorithms is that any "schedule of operations" arising by executing them is "data equivalent to a serial schedule and [preserves] the validity of the search structure." The principle that concurrent systems should be compared on the basis of their observable behaviours is, however, widely held. It has been argued to be sound specifically for database concurrency control systems in the extensive study of atomic transactions in [8]. That study employs I/O-automata, which are themselves closely related to process calculus [19]. I/O-automata do not, however, allow direct representation of systems with changing structure. On the other hand, the proofs in the present paper are related to serializability. Indeed we will show general results, couched in the π-calculus, that isolate conditions on clients and servers that guarantee atomicity of interactions between them.

The algorithms of [6] are for insertion, search, and a simple form of deletion. The insertion algorithm may require a process to hold exclusive locks on two or three nodes simultaneously. The insertion algorithm of [16] improves on this in that any process need lock at most one node at any time. Further, as observed in [16], [6] neglects to consider two cases: when the root node of the structure must be split and when a process should add a pointer but is unable to find the node where is should go (briefly, because the node that was the root when the process

was created is no longer the root). The deletion algorithm of [16] is much more elaborate than that of the earlier paper as it involves compression of the tree to avoid proliferation of sparsely occupied nodes. We give a full analysis of the insertion and search algorithms. We began with the algorithms of [6] and, in formalising and analysing them, independently rediscovered the defects and improvement published in [16]. We also carried out a rigorous analysis of the deletion and compression algorithms of [16]. In doing so we discovered an improvement to the latter, which we explain. To avoid the paper becoming over-long we omit the proofs of correctness in the cases of deletion and compression.

In the context of interactive systems, the essence of *confluence* is that the occurrence of an action will never preclude others. A theory of confluence of processes was developed in the setting of the process calculus CCS in [11]. Generalizing the idea, in [7, 14, 18] notions of *partial confluence* were introduced and studied. Their essence is that the occurrence of *certain* actions will never preclude *some* others. A key observation is that in reasoning about the behaviour of a system composed of (partial) confluent components, it may be sufficient to examine in detail only parts of the components' behaviours: from this and the fact of their (partial) confluence, it may be possible to deduce properties of the remaining behaviours. The theory of [7, 14] was used to prove the soundness of program transformation rules for concurrent object-oriented languages.

The main theoretical contribution here is an extension of the theory of partial confluence. The object-oriented systems studied in [7, 14] can be viewed as consisting of two components, Q and A, which interact in a question–answer fashion, with possibly many questions outstanding at any moment. An important property is that on accepting a question from Q, A immediately assumes a state in which the answer to that question is determined, up to behavioural equivalence. The B^{link}-system does not enjoy this property: determination of the result of an operation may involve a state change affecting subsequent operations. The extension of the theory of partial confluence to encompass systems such as the B^{link}-system is quite complicated. It covers the case when in response to an operation request, A may perform at most one state-changing internal action (in the case of the B^{link}-system, representing commitment of an operation). The main result shows that under certain natural conditions on A (and mild conditions on Q), a system composed from Q and A is indistinguishable from one composed from Q and a sequential part of A. This is the key to the proof of the correctness of the operations on the B^{link}-structure. We use it to show that the B^{link}-system is indistinguishable from the part of it in which at most one operation is in progress at any moment, that part being easily understood. A theme of the analysis is the use of the theory to reduce the complexity of the systems that must be examined in detail.

A summary of the paper follows. Section 2 contains background material on the process calculus. In Section 3 the model of the B^{link}-system is given and explained. Section 4 begins with the theory of partial confluence, stating and proving the main results. It continues with the analysis of the search and insertion algorithms and concludes with a brief discussion of alternative models of the system. In Section 5 algorithms for deleting from and compressing the data structure are considered and an improved algorithm for compression explained, with further details in the Appendix.

2. BACKGROUND

Process calculi are general theories of concurrent systems. The calculus of interest here is an extension of the π-calculus of [13], which is itself a descendant of the process calculus CCS [10, 11]. Before presenting the calculus we outline its salient features.

The calculus has a small but expressive language for describing systems. Terms of the language are interpreted as labelled transition systems whose nodes represent system states and whose arrows carry information pertinent to state change. An equivalence on transition systems induces a relation capturing when terms express systems having the same observable behaviour. The basic entities of the π-calculus are *names*. They can be thought of as names of communication links between components of systems. Components use names to interact, and by passing names in interactions, components can pass to one another the ability to interact with other components. More concretely, names can be thought of as pointers. Features of the π-calculus central to its success as a theory of pointers and of name-passing in general are its treatment of the scoping of names and the creation of names. Descriptions can be further structured by categorizing names according to the ways they can be used, making the descriptions clearer and providing information useful in reasoning with them. Type systems that achieve this are studied in many papers, for instance [12, 15, 17]. The calculus extends the π-calculus with communicable data other than names. The data part can be tailored to the application at hand; here we have integers, tuples, and variants (tagged values). To be able to categorize names as mentioned above we introduce a simple system of types for data. Assuming a set of labels (ranged over by ℓ), the types (ranged over by σ) are given by

$$\sigma ::= \mathsf{int} \mid \langle \sigma_1, ..., \sigma_n \rangle \mid [\ell_1 \_ \sigma_1, ..., \ell_n \_ \sigma_n] \mid \updownarrow \sigma \mid X \mid \mu X.\sigma.$$

In $\mu X.\sigma$, each occurrence of X in σ must be guarded by $\updownarrow$. Type equality is the smallest congruence such that $\mu X.\sigma = \sigma\{\mu X.\sigma/X\}$. We identify equal types.

We assume an infinite set of names (ranged over by x). The data values (ranged over by v) are given by

$$v ::= k \mid \langle v_1, ..., v_n \rangle \mid \ell \_ v \mid x.$$

Values are typed in a context, that is, a partial function from names to types. An integer constant k has type int. A tuple $\langle v_1, ..., v_n \rangle$ of values has type $\langle \sigma_1, ..., \sigma_n \rangle$ if each v_i has type σ_i. In the variant type $[\ell_1 \_ \sigma_1, ..., \ell_n \_ \sigma_n]$ the labels are pairwise distinct; the values of this type are those $\ell_i \_ v$ with v of type σ_i. If according to the context the name x has type $\updownarrow \sigma$, then x can be used to send and to receive values of type σ. The additional type constructs, X and $\mu X.\sigma$, allow recursive types. For instance if x is of type $\mu X. \updownarrow \langle \mathsf{int}, \updownarrow X \rangle$, then x can be used to send and to receive pairs consisting of an integer and a name that can be used to send and to receive names of the same type as x. We use b to range over Boolean expressions. We do

not specify a rigid syntax for these, but we assume that the only operation on names is equality. We use z to range over the patterns, that is, the values given by

$$z ::= x \mid \langle z_1, ..., z_n \rangle$$

in which no name occurs more than once. The agents (ranged over by P, Q) are given by

$$P ::= \sum_{j \in J} \pi_j . P_j \mid P|Q \mid (\nu x : \sigma)\, P \mid K\langle v_1, ..., v_n \rangle \mid$$
$$\mathsf{cond}(b_1 \rhd P_1, ..., b_n \rhd P_n) \mid \mathsf{case}(v)[\ell_1 - z_1 : P_1, ..., \ell_n - z_n : P_n],$$

where the prefixes are given by

$$\pi ::= x(z) \mid \bar{x}(v) \mid \tau.$$

Before showing how agents are interpreted as labelled transition systems we explain their meanings informally. The agent $\Sigma_{j \in J} \pi_j . P_j$, a finite sum, represents a choice of the prefixed summands. The prefixed agents are of three kinds. An input-prefixed agent $x(z).P$ is able to receive via the name x any value v of the appropriate type and then continue as $P\{v/z\}$, that is P with v substituted for z. An output-prefixed agent $\bar{x}(v).P$ is able to send the value v via x and continue as P. A silent-prefixed agent $\tau.P$ is able to evolve autonomously and invisibly to P. We write $\mathbf{0}$ for the empty summation; it is an agent with no capabilities. The sublanguage in which all sums have cardinality 1 or 0 is akin to a small programming language. We use sums of two or more terms only to express the observable behaviour of systems.

The agent $P \mid Q$ describes the concurrent composition of P and Q: the component agents can proceed independently and can also interact with one another using shared names. In $(\nu x)\, P$ the scope of the name x is restricted to P: components of P can use it to interact with one another but not with P's environment; however, the scope of x may change by its being sent in a communication. In $K\langle v_1, ..., v_n \rangle$, K is an agent constant with an associated definition $K(x_1, ..., x_n) \stackrel{\text{df}}{=} P$ where the pairwise distinct names $x_1, ..., x_n$ include all those occurring free in P. The agent $K\langle v_1, ..., v_n \rangle$ behaves as $P\{v_1 \cdots v_n / x_1 \cdots x_n\}$. The agent $\mathsf{cond}(b_1 \rhd P_1, ..., b_n \rhd P_n)$ is a nested conditional with Boolean expressions b_i guarding the alternatives P_i. The agent $\mathsf{case}(v)[\ell_1 - z_1 : P_1, ..., \ell_n - z_n : P_n]$ allows case analysis on the for of a value of a variant type: $\mathsf{case}(\ell_i - v)[\ell_1 - z_1 : P_1, ..., \ell_n - z_n : P_n]$ behaves as $P_i\{v/z_i\}$. A convenient derived form is the replicator $!P$ defined by $!P(x_1, ..., x_n) \stackrel{\text{df}}{=} (P \mid !P\langle x_1, ..., x_n \rangle)$, which may generate arbitrarily many instances of P.

This informal account is made precise as follows. We consider only agents that are well typed, i.e., in which the use of names accords with that determined by the ambient context. We write Γ for the ambient context, a function from names to types. A family of rules is given for inferring judgments of the forms $\Gamma \vdash v : \sigma$ (in context Γ, value v has type σ) and $\Gamma \vdash P$ (in context Γ, agent P is well typed). We give just one of the straightforward rules: if $\Gamma \vdash P$, $\Gamma(x) = \updownarrow \sigma$ and $\Gamma \vdash v : \sigma$, then $\Gamma \vdash \bar{x}(v).P$. In $x(z).P$ the occurrences of names in z are binding with scope P. In

$(\nu x:\sigma)\,P$ the occurrence of x is binding with scope P. (We often elide the type annotation in restrictions.) We write $\mathrm{fn}(P)$ for the set of free names of P, i.e., those names occurring in P not within the scope of a binding occurrence. We identify agents that differ only by change of bound names.

The actions (ranged over by α) are given by

$$\alpha ::= xv \mid \bar{x}(\nu\tilde{x})\,v \mid \tau,$$

where in the second form the pairwise distinct names in the tuple $\tilde{x} = x_1 \cdots x_n$ occur in v. If the tuple $\tilde{x}$ is empty, we write simply $\bar{x}v$. We write *Act* for the set of actions. The transition relation defined by the rules below, have the following interpretations. First, $P \xrightarrow{xv} Q$ means that P can receive the value v via x and thereby evolve into Q. Second, $P \xrightarrow{\bar{x}(\nu\tilde{x})\,v} Q$ means that P can send the value v via x and thereby evolve into Q—the scopes of the names in $\tilde{x}$ are enlarged by the transition. And third, $P \xrightarrow{\tau} Q$ means that P can evolve invisibly into Q. The *subject* of an action α is defined as follows: if α is xv then $\mathsf{subj}(\alpha) = x$, if α is $\bar{x}(\nu\tilde{x})\,v$ then $\mathsf{subj}(\alpha) = \bar{x}$, and $\mathsf{subj}(\tau) = \tau$. The set $\mathrm{bn}(\alpha)$ of bound names of α is the set containing the names in $\tilde{x}$ if α is $\bar{x}(\nu\tilde{x})\,v$ and $\varnothing$ otherwise. The rules are as follows where we write $b \rightsquigarrow c$ to express that the Boolean expression b evaluates to c (evaluation is a partial function) and where we elide the symmetric forms of the fourth and fifth rules:

1. $\cdots + x(z).P + \cdots \xrightarrow{xv} P\{v/z\}$ if $\Gamma(x) = \updownarrow\sigma$ and $\Gamma \vdash v : \sigma$,
2. $\cdots + \bar{x}(v).P + \cdots \xrightarrow{\bar{x}v} P$,
3. $\cdots + \tau.P + \cdots \xrightarrow{\tau} P$,
4. if $P \xrightarrow{\alpha} P'$ then $P \mid Q \xrightarrow{\alpha} P' \mid Q$ provided $\mathrm{bn}(\alpha) \cap \mathrm{fn}(Q) = \varnothing$,
5. if $P \xrightarrow{\bar{x}(\nu\tilde{x})\,v} P'$ and $Q \xrightarrow{xv} Q'$ then $P \mid Q \xrightarrow{\tau} (\nu\tilde{x})(P' \mid Q')$ provided $\tilde{x} \cap \mathrm{fn}(Q) = \varnothing$,
6. if $P \xrightarrow{\alpha} P'$ then $(\nu x)\,P \xrightarrow{\alpha} (\nu x)\,P'$ provided x does not occur in α,
7. if $P \xrightarrow{\bar{x}(\nu\tilde{x})\,v} P'$ then $(\nu y)\,P \xrightarrow{\bar{x}(\nu\tilde{x})\,v} P'$ provided y occurs in $v - (\tilde{x} \cup \{x\})$,
8. if $K(\tilde{x}) \stackrel{\mathrm{df}}{=} P$ and $P\{\tilde{v}/\tilde{x}\} \xrightarrow{\alpha} P'$, then $K\langle\tilde{v}\rangle \xrightarrow{\alpha} P'$,
9. if $b_j \rightsquigarrow \mathsf{false}$ for $j < i$, $b_i \rightsquigarrow \mathsf{true}$ and $P_i \xrightarrow{\alpha} P$, then $\mathsf{cond}(b_1 \rhd P_1, \ldots, b_n \rhd P_n) \xrightarrow{\alpha} P$,
10. if $P_i\{v/z_i\} \xrightarrow{\alpha} P$ then $\mathsf{case}(\ell_i\text{–}v)[\ell_1\text{–}z_1 : P_1, \ldots, \ell_n\text{–}z_n : P_n] \xrightarrow{\alpha} P$.

Here is an example to illustrate the rules:

$$\begin{aligned}
&(\nu x)(P \mid x(y,z).\mathsf{cond}(y<5 \rhd \bar{z}(0).Q,\ y \geqslant 5 \rhd \bar{z}(1).Q') \mid (\nu w)\,\bar{x}(\langle 3, w\rangle).w(u).R) \\
&\quad \xrightarrow{\tau} (\nu x)(P \mid (\nu w)(\mathsf{cond}(3<5 \rhd \bar{w}(0).Q,\ 3 \geqslant 5 \rhd \bar{w}(1).Q') \mid w(u).R)) \\
&\quad \xrightarrow{\tau} (\nu x)(P \mid (\nu w)(Q \mid R\{0/u\})).
\end{aligned}$$

Note how the scope of w grows in the first transition. The combination $(\nu w)\,\bar{x}(\langle 3, w\rangle)$ expresses the sending via x of 3 and a *fresh* name w. Further, if $w \notin \mathrm{fn}(Q) \cup \mathrm{fn}(R)$ then (νw) can be removed from the last agent as it is semantically insignificant.

Semantic claims such as this are justified on the basis of a precise account of behavioural equivalence of agents. Many notions of behavioural equivalence have been considered. Here we employ a well-studied equivalence called *branching bisimilarity* [4]. An important feature is that it abstracts from silent actions. It is one of the most demanding reasonable notions of equivalence. We adopt it here first as it enables us to give a very precise description of the behaviour of the system we consider, and second because the techniques for reasoning with it are powerful. A useful piece of notation: we write $\Rightarrow$ for the reflexive and transitive closure of $\xrightarrow{\tau}$.

DEFINITION 2.1. Branching bisimilarity, $\simeq$, is the largest *branching bisimulation*, i.e., symmetric relation $\mathscr{B}$ on agents such that if $P\mathscr{B}Q$ then for all actions α with $\mathrm{bn}(\alpha) \cap \mathrm{fn}(P, Q) = \varnothing$, if $P \xrightarrow{\alpha} P'$ then (1) for some Q'', Q', $Q \Rightarrow Q'' \xrightarrow{\alpha} Q'$, $P\mathscr{B}Q''$, and $P'\mathscr{B}Q'$, or (2) $\alpha = \tau$ and $P'\mathscr{B}Q$.

Thus any transition of either of a pair of branching-bisimilar agents can be matched by a computation of the other with the same visible content and which respects the branching structure of the agents. The restriction that any bound names in an action should not be free in either agent ensures that they are required to have the same behaviour only when bound names in output actions are fresh to both. To show that a pair of agents are branching bisimilar, it suffices to find a branching bisimulation relating them. Branching bisimilarity is an equivalence relation and is preserved by all the operators except input-prefix. These results can be shown using standard techniques.

3. DATA STRUCTURE AND OPERATIONS

This section contains the process-calculus descriptions of the B^{link}-tree and the concurrent insertion and search algorithms. Salient points in [6, 16] are that the actions of writing to and reading from the disk are assumed to be atomic, that in order to write to a disk page a process must hold an exclusive lock on it, and that no process is prevented from reading a page because another process holds a lock on it. As mentioned in the Introduction we express both the operations and the data as (active) agents. We discuss alternative representations in Section 4.3. We begin with a brief informal description of the data structure.

A B^{link}-tree indexes a database by storing in its leaves pairs $\langle k, b \rangle$ with k an integer key associated with a record and b a pointer to the record. All of its leaves are at the same distance from the root. Each of its nonleaf nodes has j keys and j pointers where $2 \leqslant j \leqslant 2m+1$ if the node is the root and $m+1 \leqslant j \leqslant 2m+1$ otherwise (for some tree parameter m). A node stores its keys in ascending order. It is intended that a node's largest key, its *high key*, is the largest key in the subtree rooted at the node. All but the last pointer of a nonleaf node point to children of the node. All but the last of a leaf's pointers point to records of the database. The last pointer of a node or leaf, its *link*, points to its right neighbour at the same level of the tree, if it exists. The purpose of links is to provide additional paths through

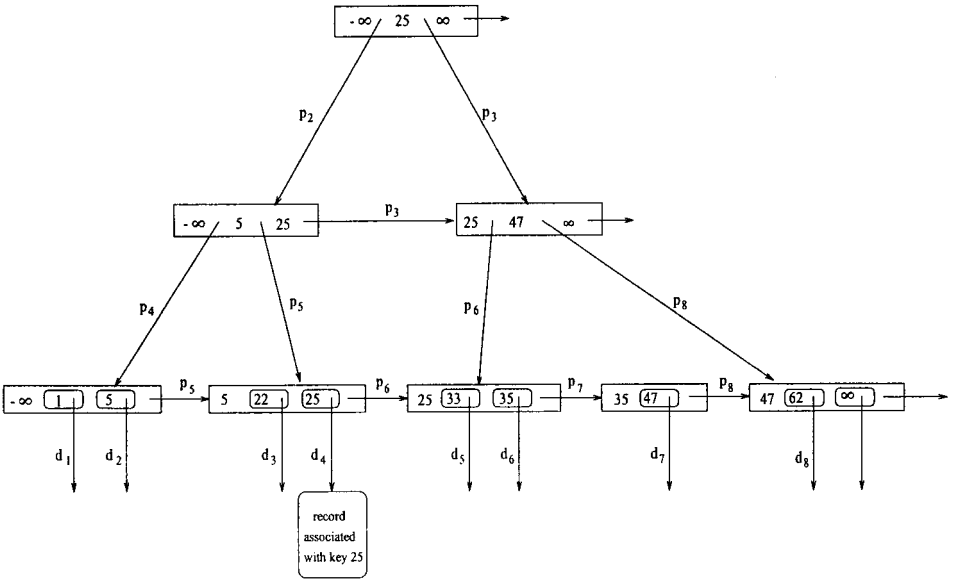


FIG. 1. A B^{link}-tree.

the structure. The rightmost node at each level has high key ∞ and link *nil*. If a nonleaf node has keys $\tilde{k} = k_1, ..., k_j$ and pointers $\tilde{p} = p_1, ..., p_j$, then an intended invariant is that for $i < j$, pointer p_i points to a subtree whose leaves contain all keys k with $k_i < k \leqslant k_{i+1}$. An example of a B^{link}-tree is given in Fig. 1.

We now give the process-calculus description of the B^{link}-tree and the operations on it. In its initial state it will be an agent of the form $(v\tilde{x})(S_0 \mid I_0 \mid T_0)$ where S_0 and I_0 represent the search and insertion operations, respectively, and T_0 the tree in its initial state. It will have free names s and i via which the operations may be requested.

We focus first on the types of data that may be passed between the component agents. We stipulate that the labels (of variants) are:

link, *nonlink*, *datum*, *done*, *split*, *search*, *insert*, *add*.

These labels will be used to tag data that processes communicate, and the names are suggestive of the purposes of the various communications, as will become clear. We let D be the type of names representing pointers to database records stored in the leaves of the tree. We introduce synonyms for two mutually-recursive types:

$$\mathsf{P} \equiv [\mathit{search}\_\langle \mathsf{int}, \updownarrow \mathsf{R} \rangle, \mathit{insert}\_\langle \mathsf{int}, \mathsf{D}, \updownarrow \mathsf{R} \rangle, \mathit{add}\_\langle \mathsf{int}, \updownarrow \mathsf{P}, \updownarrow \mathsf{R} \rangle]$$

$$\mathsf{R} \equiv [\mathit{link}\_\updownarrow \mathsf{P}, \mathit{nonlink}\_\updownarrow \mathsf{P}, \mathit{datum}\_\mathsf{D}, \mathit{done}\_\langle\ \rangle, \mathit{split}\_\langle \updownarrow \mathsf{P}, \mathsf{int} \rangle].$$

TABLE 1

Names	Type
$p, q, p', q', p_1, q_1, \ldots$	$\updownarrow \mathsf{P}$
r	$\updownarrow \mathsf{R}$
z	P
y	R
b	D
$k, k_1, \ldots,$	int
s	$\updownarrow \langle \mathsf{int}, \updownarrow \mathsf{D} \rangle$
i	$\updownarrow \langle \mathsf{int}, \mathsf{D}, \updownarrow \langle\ \rangle \rangle$
a_s	$\updownarrow \mathsf{D}$
a_i	$\updownarrow \langle\ \rangle$
$next$	$\updownarrow \langle \updownarrow \mathsf{P}, \updownarrow \updownarrow \mathsf{P} \rangle$
get, put, n	$\updownarrow \updownarrow \mathsf{P}$

A name of type $\updownarrow \mathsf{P}$ will represent a pointer to a tree-node; it can be thought of as the name of the node. A name of type $\updownarrow \mathsf{R}$ will be used in interrogating a node. Table 1 summarises the typing context for the definitions to follow.

The B^{link}-tree in its initial state is represented by the agent

$$T_0(get, next) \stackrel{\text{df}}{=} (vp, p', put)(\mathsf{ROOT}\ \langle p, \langle -\infty, \infty \rangle, \langle p', nil \rangle, put \rangle \\ \mid \mathsf{LEAF}\ \langle p', \langle -\infty, \infty \rangle, \langle nil \rangle, nil \rangle \\ \mid \mathsf{STORE}\ \langle \langle p \rangle, get, put, next \rangle).$$

The name *nil* of type $\updownarrow \mathsf{P}$ or D represents a nil pointer, ROOT represents the root of the tree, and LEAF the only leaf. The role of STORE is to record the name of each node that is the root of the structure at some point in its evolution. The interface between the tree agent and the environment is the names *get* and *next*, via which the environment can acquire the name of the current root and the names of former roots of the tree, respectively. This is explained in detail later.

We first define an agent $\mathsf{NODE}\ \langle p, \tilde{k}, \tilde{p} \rangle$ representing a nonroot, nonleaf node named p storing keys $\tilde{k} = k_1 \cdots k_j$ and pointers $\tilde{p} = p_1 \cdots p_j$ to node. In this and later definitions, $\|_{1 \leqslant i \leqslant \ell}\, b(i) \rhd P(i)$ abbreviates $b(1) \rhd P_1, \ldots, b(\ell) \rhd P_\ell$. The definition is

$\mathsf{NODE}(p, \tilde{k}, \tilde{p})$

$$\stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[search\_\langle k, r \rangle : \\ \mathsf{cond}(k > k_j \qquad \rhd \bar{r}(link\_p_j).N, \\ \|_{1 \leqslant h \leqslant j-1}\, k_{h+1} \geqslant k > k_h \rhd \bar{r}(nonlink\_p_h).N), \\ add\_\langle k, q, r \rangle : \\ \mathsf{cond}(k > k_j \qquad \rhd \bar{r}(link\_p_j).N, \\ \|_{1 \leqslant h \leqslant j-1}\, k = k_h \rhd \bar{r}(done\_\langle\ \rangle).N_h, \\ notfull \qquad \rhd \bar{r}(done\_\langle\ \rangle).N', \\ full \qquad \rhd (vp')\, \bar{r}(split\_\langle p', k_{m+1} \rangle).(N_1 \mid N_2))],$$

where $N = \text{NODE}\langle p, \tilde{k}, \tilde{p}\rangle$ and N_h, N_1, and N_2 are defined below. In its quiescent state a node can accept via its name p a search request and an add request. The former contains an integer k (to search for) and a name r (via which to return the result of the search) tagged with the label *search* to specify the operation. In response to a search request the node returns the appropriate pointer, labelled to indicate whether or not it is its link, and resumes its quiescent state, N.

An add request contains a pair k, q (to be added to the node) and a name r (via which to return a response) tagged with the label *add*. If k is larger than the high key of the node, the link is returned. If k is already present the associated pointer is updated.

$$N_h = \text{NODE}\langle p, \tilde{k}, p_1 \cdots p_{h-1}\, q\, p_{h+1} \cdots p_j\rangle.$$

The Boolean expression *notfull* is true if the node can accommodate the pair, i.e., if $j \leqslant 2m$. Then

$$N' = \text{NODE}\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_j, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_j\rangle,$$

where $k_h < k < k_{h+1}$. The last possibility is that the node is full; i.e., $j = 2m+1$. In this case the node is split. More precisely a node is created and the data are shared between the two. There are two cases depending on whether the pair k, q is added to the new node or the existing node. Let h be such that $k_h < k < k_{h+1}$. Then

1. if $k_{m+1} \leqslant k_h$ then

$$N_1 = \text{NODE}\langle p, k_1 \cdots k_{m+1}, p_1 \cdots p_m\, p'\rangle$$

$$N_2 = \text{NODE}\langle p', k_{m+1} \cdots k_h\, k\, k_{h+1} \cdots k_{2m+1}, p_{m+1} \cdots p_h\, q\, p_{h+1} \cdots p_{2m+1}\rangle;$$

2. if $k_{h+1} \leqslant k_{m+1}$ then

$$N_1 = \text{NODE}\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_{m+1}, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_m, p'\rangle$$

$$N_2 = \text{NODE}\langle p', k_{m+1} \cdots k_{2m+1}, p_{m+1} \cdots p_{2m+1}\rangle.$$

In each case via r is returned the pair $\langle p', k_{m+1}\rangle$ consisting of the name of the new node and its smallest key k_{m+1} (which also becomes the largest key of the node which was split), tagged with the label *split*. The recipient of this pair, the agent responsible for initiating the add, will add it to a node one level up in the tree (see below). Duplication of the key k_{m+1} facilitates the analysis. In Section 4.3 we show that it is not necessary for correctness of the algorithms.

The agent $\text{LEAF}\langle p, \tilde{k}, \tilde{b}, q\rangle$ representing a leaf named p storing keys $\tilde{k} = k_1 \cdots k_j$, pointers $\tilde{b} = b_2 \cdots b_j$ to database records, and link q is defined by

$\mathsf{LEAF}(p, \tilde{k}, \tilde{b}, q)$

$$\stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[\,search\_\langle k, r\rangle :$$
$$\mathsf{cond}(k > k_j \quad \triangleright\ \bar{r}(link\_q).L,$$
$$\|_{1 \leqslant h \leqslant j-1}\, k = k_h \ \triangleright\ \bar{r}(datum\_b_h).L,$$
$$k \notin \tilde{k} \quad \triangleright\ \bar{r}(datum\_nil).L),$$

$$insert\_\langle k, b, r\rangle:$$
$$\mathsf{cond}(k > k_j \quad \triangleright\ \bar{r}(link\_q).L,$$
$$\|_{1 \leqslant h \leqslant j-1}\, k = k_h \ \triangleright\ \bar{r}(done\_\langle\ \rangle).L_h,$$
$$notfull \quad \triangleright\ \bar{r}(done\_\langle\ \rangle).L',$$
$$full \quad \triangleright\ (vp')\,\bar{r}(split\_\langle p', k_{m+1}\rangle).(L_1 \mid L_2))].$$

Note that LEAF stores $j-1$ key–pointer pairs $(k_2, b_2, ..., k_j, b_j)$ and that the key k_1 has no associated pointer to a database record (in this LEAF, k_1 will be the high key of the leaf's left neighbour, and as just mentioned this simplifies the analysis). In its quiescent state a leaf can accept a search request and an insertion request. In response to the former it reacts similarly to NODE, except that if $k \leqslant k_j$ it returns, suitably tagged, the appropriate pointer to the database or *nil* (of type D) if k is absent and resumes its quiescent state: $L = \mathsf{LEAF}\ \langle p, \tilde{k}, \tilde{b}, q\rangle$. In response to a request to insert a $\langle k, b\rangle$ pair, LEAF behaves analogously to how NODE behaves in response to an add request (in the second case h ranges over 2, ..., j),

$$L_h = \mathsf{LEAF}\ \langle p, \tilde{k}, b_2 \cdots b_{h-1}\, b\, b_{h+1} \cdots b_j\rangle$$

$$L' = \mathsf{LEAF}\ \langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_j, b_2 \cdots b_h\, b\, b_{h+1} \cdots b_j\rangle,$$

where $k_h < k < k_{h+1}$. Finally, if $k_h < k < k_{h+1}$ then

1. if $k_{m+1} \leqslant k_h$ then

$$L_1 = \mathsf{LEAF}\ \langle p, k_1 \cdots k_{m+1}, b_2 \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\ \langle p', k_{m+1} \cdots k_h\, k\, k_{h+1} \cdots k_{2m+1}, b_{m+2} \cdots b_h\, b\, b_{h+1} \cdots b_{2m+1}, q\rangle,$$

2. if $k_{h+1} \leqslant k_{m+1}$ then

$$L_1 = \mathsf{LEAF}\ \langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_{m+1}, b_2 \cdots b_h\, b\, b_{h+1} \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\ \langle p', k_{m+1} \cdots k_{2m+1}, b_{m+2} \cdots b_{2m+1}, q\rangle,$$

The agent $\mathsf{ROOT}\ \langle p, \tilde{k}, \tilde{p}, put\rangle$ representing a root node named p storing keys $\tilde{k} = k_1 \cdots k_j$ and pointers $\tilde{p} = p_1 \cdots p_j$ to nodes is defined by

$$
\begin{aligned}
&\mathsf{ROOT}(p, \tilde{k}, \tilde{p}, put)\\
&\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[search\_\langle k, r\rangle :\\
&\qquad\qquad \mathsf{cond}(\|_{1\leqslant h\leqslant j} k_{h+1} \geqslant k > k_h \rhd \bar{r}(nonlink\_p_h).R),\\
&\qquad\qquad add\_\langle k, q, r\rangle :\\
&\qquad\qquad \mathsf{cond}(\|_{1\leqslant h\leqslant j} k = k_h \rhd \bar{r}(done\_\langle\ \rangle).R_h,\\
&\qquad\qquad\qquad notfull \quad \rhd \bar{r}(done\_\langle\ \rangle).R',\\
&\qquad\qquad\qquad full \quad \rhd \bar{r}(done\_\langle\ \rangle).(\nu p_0)\,\overline{put}(p_0).\\
&\qquad\qquad\qquad\qquad (\nu p')(\mathsf{NEWROOT} \mid N_1 \mid N_2))].
\end{aligned}
$$

In response to a search request ROOT reacts similarly to NODE: $R = \mathsf{ROOT}\langle p, \tilde{k}, \tilde{p}, put\rangle$. The first two alternatives in response to an add request are similar to those of NODE.

$$
\begin{aligned}
R_h &= \mathsf{ROOT}\langle p, \tilde{k}, p_1 \cdots p_{h-1}\, q\, p_{h+1} \cdots p_j\rangle,\\
R' &= \mathsf{NODE}\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_j, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_j\rangle,
\end{aligned}
$$

where $k_h < k < k_{h+1}$. In the final possibility in which the root is split there are again two cases. In both $\mathsf{NEWROOT} = \mathsf{ROOT}\langle p_0, \langle -\infty, k_{m+1}, \infty\rangle, \langle p, p', nil\rangle, put\rangle$. Also, assuming that for some h, $k_h < k < k_{h+1}$, then

1. if $k_{m+1} \leqslant k_h$ then

$$
\begin{aligned}
N_1 &= \mathsf{NODE}\langle p, k_1 \cdots k_{m+1}, p_1 \cdots p_m\, p'\rangle\\
N_2 &= \mathsf{NODE}\langle p', k_{m+1} \cdots k_h\, k\, k_{h+1} \cdots k_{2m+1}, p_{m+1} \cdots p_h\, q\, p_{h+1} \cdots p_{2m+1}\rangle,
\end{aligned}
$$

2. if $k_{h+1} \leqslant k_{m+1}$ then

$$
\begin{aligned}
N_1 &= \mathsf{NODE}\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_{m+1}, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_m\, p'\rangle\\
N_2 &= \mathsf{NODE}\langle p', k_{m+1} \cdots k_{2m+1}, p_{m+1} \cdots p_{2m+1}\rangle.
\end{aligned}
$$

Thus the root assumes node status (N_1), a node is created (N_2), and a root is created (NEWROOT). The name of the new root is put into the store via the name *put*.

The agent $\mathsf{STORE}\langle \tilde{p}, get, put, next\rangle$ responsible for recording in order of creation the names $\tilde{p} = p_1, ..., p_j$ of the current and previous roots is defined by

$$
\begin{aligned}
\mathsf{STORE}(\tilde{p}, get, put, next) \stackrel{\text{df}}{=}\ & \overline{get}(p_j).S +\\
& put(p).S' +\\
& next(p, n).\mathsf{cond}(p = p_h \rhd \bar{n}(p_{h+1}).S),
\end{aligned}
$$

where $S = \mathsf{STORE}\langle \tilde{p}, get, put, next\rangle$ and $S' = \mathsf{STORE}\langle \tilde{p}p, get, put, next\rangle$. The store can deliver via *get* the name p_j of the current root, can record the name of a new root received via *put*, and when sent via *next* the name p of a former root can return via n the name of its successor as root.

This completes the definition of the agent T_0 representing the B^{link}-tree in its initial state. We now describe the search and insertion operations.

The searcher $S_0\langle s, get\rangle$ is defined by

$$S_0(s, get) \stackrel{\text{df}}{=} !\, s(k, a_s).get(p).S\langle k, p, a_s\rangle$$
$$S(k, p, a_s) \stackrel{\text{df}}{=} (vr)\ \bar{p}(search\_\langle k, r\rangle).r(y).\mathsf{case}(y)[link\_p' : S\langle k, p', a_s\rangle,\ nonlink\_p' : S\langle k, p', a_s\rangle,\ datum\_b : \overline{a_s}(b).\mathbf{0}].$$

The agent $S_0\langle s, get\rangle$ may repeatedly spin off searches when supplied via s with an integer k to search for and a name a_s via which to return the result of the search. For instance

$$S_0\langle s, get\rangle \xrightarrow{s\langle k, a_s\rangle} \xrightarrow{get\ p} S\langle k, p, a_s\rangle \mid S_0\langle s, get\rangle$$
$$\xrightarrow{s\langle k', a_s'\rangle} \xrightarrow{get\ p'} S\langle k', p', a_s'\rangle \mid S\langle k, p, a_s\rangle \mid S_0\langle s, get\rangle.$$

On initiation of a search, the searcher reads from the STORE via *get* the name of the root. It then traces a path through the structure until it reaches some LEAF that synchronizes with it by performing an action $\bar{r}\, datum\_b$ returning the result b of the search (which may be *nil* indicating that the key is absent). That result is emitted via a_s and the searcher becomes inactive.

The inserter $I_0\langle i, get, next\rangle$ is defined by:

$$I_0(i, get, next) \stackrel{\text{df}}{=} !\, i(k, b, a_i).get(p).Down\langle k, b, a_i, p, \langle\ \rangle\rangle$$

$$Down(k, b, a_i, p, \tilde{q}) \stackrel{\text{df}}{=} (vr)\ \bar{p}(search\_\langle k, r\rangle).r(y).$$
$$\mathsf{cond}(\tilde{q} = \langle\ \rangle \triangleright \mathsf{case}(y)[link\_p' : Down\langle k, b, a_i, p', \langle p\rangle\rangle,\ nonlink\_p' : Down\langle k, b, a_i, p', \langle p\rangle\rangle],$$
$$\tilde{q} \neq \langle\ \rangle \triangleright \mathsf{case}(y)[link\_p' : Down\langle k, b, a_i, p', \tilde{q}\rangle,\ nonlink\_p' : Down\langle k, b, a_i, p', p\tilde{q}\rangle,\ datum\_b' : Insert\langle k, b, a_i, p, \tilde{q}\rangle])$$

$$Insert(k, b, a_i, p, q\tilde{q}) \stackrel{\text{df}}{=} (vr)\ \bar{p}(insert\_\langle k, b, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_p' : Insert\langle k, b, a_i, p', q\tilde{q}\rangle,\ done\_\langle\ \rangle : \overline{a_i}.\mathbf{0},\ split\_\langle p', k'\rangle : \overline{a_i}.Up\langle k', p', q, \tilde{q}\rangle]$$

$$Up(k, p, q, q_0\tilde{q}) \stackrel{\text{df}}{=} (vr)\ \bar{q}(add\_\langle k, p, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_q' : Up\langle k, p, q', q_0\tilde{q}\rangle,\ done\_\langle\ \rangle : \mathbf{0},\ split\_\langle p', k'\rangle : \mathsf{cond}(\tilde{q} \neq \langle\ \rangle \triangleright Up\langle k', p', q_0, \tilde{q}\rangle,\ \tilde{q} = \langle\ \rangle \triangleright Up'\langle k', p', q_0, \langle q_0\rangle\rangle)]$$

$$Up'(k, p, q, \langle q_0\rangle) \stackrel{\text{df}}{=} (vr)\ \bar{q}(add\_\langle k, p, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_q' : Up'\langle k, p, q', \langle q_0\rangle\rangle,\ done\_\langle\ \rangle : \mathbf{0},\ split\_\langle p', k'\rangle : (vn)\ \overline{next}(\langle q_0, n\rangle).n(q_1).Up'\langle k', p', q_1, \langle q_1\rangle\rangle].$$

The replicator $I_0\langle i, get, next\rangle$ can repeatedly spin off inserters when supplied via i with a pair k, b to insert and a name a_i via which to send confirmation that the insertion has been done. Note that we write $\overline{a_i}$ for $\overline{a_i}(\langle\,\rangle)$. The inserter obtains the name of the root from the STORE and searches until the appropriate leaf is reached. Note that its starting point and the names of the rightmost nodes in the path followed are recorded in the last parameter; thus the last name in that parameter is the name of the node which was the root when the inserter was created. An insertion within a leaf may result in it splitting. The inserter is informed of this by being sent a pair $\langle p', k'\rangle$ suitably tagged. In such a case the continuation agent Up is responsible for inserting the pair $\langle p', k'\rangle$ in a node one level higher in the tree. This process may be repeated by Up in several levels of the tree. This is the reason that the names $\tilde{q}$ of the rightmost nodes visited are recorded during the searching phase. It is possible that $\tilde{q}$ may become empty although an insertion is required at a higher level of the tree: new levels may have been created after the inserter began its task. If this happens the inserter Up' queries the STORE via $next$ to obtain the name of the leftmost node at the level above. This process too may be repeated. Note that there may be two pointers to a node: one from a node at the level above it in the tree and one from its left neighbour. One might think that as far as correctness of the operations is concerned, an insertion in a leaf of the structure does not require that any updates be performed in higher levels of the tree, that is, that the Up phase of an insertion could be omitted. Our analysis will show that this is indeed the case. The Up phase is, however, important for efficiency. (In the model, omitting the Up phase means replacing $Up\langle\cdots\rangle$ by $\mathbf{0}$.)

As outlined earlier, the system consisting of the structure and the operations is represented by the agent

$$P_0(s, i) \stackrel{\text{df}}{=} (vget, next)(S_0\langle s, get\rangle \mid I_0\langle i, get, next\rangle \mid T_0\langle get, next\rangle).$$

For ease of reference, we collect the agent definitions below. We then proceed to analyse the model.

$$\begin{aligned} T_0(get, next) \stackrel{\text{df}}{=} (vp, p', put)(&\mathsf{ROOT}\,\langle p, \langle -\infty, \infty\rangle, \langle p', nil\rangle, put\rangle \\ &\mid \mathsf{LEAF}\,\langle p', \langle -\infty, \infty\rangle, \langle nil\rangle, nil\rangle \\ &\mid \mathsf{STORE}\,\langle\langle p\rangle, get, put, next\rangle). \end{aligned}$$

$$\begin{aligned} &\mathsf{NODE}(p, \tilde{k}, \tilde{p}) \\ &\quad\stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[search\_\langle k, r\rangle : \\ &\qquad\mathsf{cond}(k > k_j \qquad\qquad\qquad\qquad \triangleright\ \bar{r}(link\_p_j).N, \\ &\qquad\qquad \|_{1\leqslant h\leqslant j-1}\, k_{h+1} \geqslant k > k_h \ \triangleright\ \bar{r}(nonlink\_p_h).N), \\ &\qquad add\_\langle k, q, r\rangle : \\ &\qquad\mathsf{cond}(k > k_j \qquad\qquad\ \triangleright\ \bar{r}(link\_p_j).N, \\ &\qquad\qquad \|_{1\leqslant h\leqslant j-1}\, k = k_h \ \triangleright\ \bar{r}(done\_\langle\,\rangle).N, \\ &\qquad\qquad notfull \qquad\qquad \triangleright\ \bar{r}(done\_\langle\,\rangle).N', \\ &\qquad\qquad full \qquad\qquad\ \triangleright\ (vp')\,\bar{r}(split\_\langle p', k_{m+1}\rangle).(N_1 \mid N_2))], \end{aligned}$$

where

$$N = \mathsf{NODE}\ \langle p, \tilde{k}, \tilde{p}\rangle$$

$$N_h = \mathsf{NODE}\ \langle p, \tilde{k}, p_1 \cdots p_{h-1}\ q\ p_{h+1} \cdots p_j\rangle$$

$$N' = \mathsf{NODE}\ \langle p, k_1 \cdots k_h\ k\ k_{h+1} \cdots k_j, p_1 \cdots p_h\ q\ p_{h+1} \cdots p_j\rangle \quad \text{where } k_h < k < k_{h+1},$$

and where if h is such that $k_{m+1} \leqslant k_h < k < k_{h+1}$ then

$$N_1 = \mathsf{NODE}\ \langle p, k_1 \cdots k_{m+1}, p_1 \cdots p_m\ p'\rangle$$

$$N_2 = \mathsf{NODE}\ \langle p', k_{m+1} \cdots k_h\ k\ k_{h+1} \cdots k_{2m+1}, p_{m+1} \cdots p_h\ q\ p_{h+1} \cdots p_{2m+1}\rangle,$$

and if h is such that $k_{m+1} \leqslant k_h < k < k_{h+1} \leqslant k_{m+1}$ then

$$N_1 = \mathsf{NODE}\ \langle p, k_1 \cdots k_h\ k\ k_{h+1} \cdots k_{m+1}, p_1 \cdots p_h\ q\ p_{h+1} \cdots p_m, p'\rangle$$

$$N_2 = \mathsf{NODE}\ \langle p', k_{m+1} \cdots k_{2m+1}, p_{m+1} \cdots p_{2m+1}\rangle.$$

$$\begin{aligned}
&\mathsf{LEAF}(p, \tilde{k}, \tilde{b}, q)\\
&\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[\mathit{search}_\langle k, r\rangle:\\
&\qquad\qquad \mathsf{cond}(k > k_j \qquad\qquad \triangleright\ \bar{r}(\mathit{link}_q).L,\\
&\qquad\qquad\qquad \|_{1 \leqslant h \leqslant j-1}\, k = k_h \triangleright\ \bar{r}(\mathit{datum}_b_h).L,\\
&\qquad\qquad\qquad k \notin \tilde{k} \qquad\qquad \triangleright\ \bar{r}(\mathit{datum}_\mathit{nil}).L),\\
&\qquad\qquad \mathit{insert}_\langle k, b, r\rangle:\\
&\qquad\qquad \mathsf{cond}(k > k_j \qquad\qquad \triangleright\ \bar{r}(\mathit{link}_q).L,\\
&\qquad\qquad\qquad \|_{1 \leqslant h \leqslant j-1}\, k = k_h \triangleright\ \bar{r}(\mathit{done}_\langle\ \rangle).L_h,\\
&\qquad\qquad\qquad \mathit{notfull} \qquad\qquad \triangleright\ \bar{r}(\mathit{done}_\langle\ \rangle).L',\\
&\qquad\qquad\qquad \mathit{full} \qquad\qquad \triangleright\ (vp')\,\bar{r}(\mathit{split}_\langle p', k_{m+1}\rangle).(L_1 \mid L_2))],
\end{aligned}$$

where

$$L = \mathsf{LEAF}\ \langle p, \tilde{k}, \tilde{b}, q\rangle$$

$$L_h = \mathsf{LEAF}\ \langle p, \tilde{k}, b_2 \cdots b_{h-1}\ b\ b_{h+1} \cdots b_j\rangle$$

$$L' = \mathsf{LEAF}\ \langle p, k_1 \cdots k_h\ k\ k_{h+1} \cdots k_j, b_2 \cdots b_h\ b\ b_{h+1} \cdots b_j\rangle, \quad \text{where and } k_h < k < k_{h+1}$$

and if $k_{m+1} \leqslant k_h < k < k_{h+1}$ then

$$L_1 = \mathsf{LEAF}\ \langle p, k_1 \cdots k_{m+1}, b_2 \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\ \langle p', k_{m+1} \cdots k_h\ k\ k_{h+1} \cdots k_{2m+1}, b_{m+2} \cdots b_h\ b\ b_{h+1} \cdots b_{2m+1}, q\rangle,$$

and if $k_h < k < k_{h+1} \leqslant k_{m+1}$ then

$$L_1 = \mathsf{LEAF}\ \langle p, k_1 \cdots k_h\ k\ k_{h+1} \cdots k_{m+1}, b_2 \cdots b_h\ b\ b_{h+1} \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\ \langle p', k_{m+1} \cdots k_{2m+1}, b_{m+2} \cdots b_{2m+1}, q\rangle.$$

$$\begin{aligned}
&\mathsf{ROOT}(p, \tilde{k}, \tilde{p}, put)\\
&\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[search_\langle k, r\rangle :\\
&\qquad\qquad \mathsf{cond}(\|_{1\leqslant h\leqslant j} k_{h+1} \geqslant k > k_h \rhd \bar{r}(nonlink_p_h).R),\\
&\qquad\qquad add_\langle k, q, r\rangle :\\
&\qquad\qquad \mathsf{cond}(\|_{1\leqslant h\leqslant j} k = k_h \rhd \bar{r}(done_\langle\ \rangle).R_h,\\
&\qquad\qquad\qquad notfull \rhd \bar{r}(done_\langle\ \rangle).R',\\
&\qquad\qquad\qquad full \rhd \bar{r}(done_\langle\ \rangle).(vp_0)\,\overline{put}(p_0).\\
&\qquad\qquad\qquad\qquad (vp')(\mathsf{NEWROOT} \mid N_1 \mid N_2))],
\end{aligned}$$

where

$$\begin{aligned}
R &= \mathsf{ROOT}\,\langle p, \tilde{k}, \tilde{p}, put\rangle\\
R_h &= \mathsf{ROOT}\,\langle p, \tilde{k}, p_1 \cdots p_{h-1}\, q\, p_{h+1} \cdots p_j\rangle,\\
R' &= \mathsf{NODE}\,\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_j, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_j\rangle,\\
&\qquad \text{where} \quad k_h < k < k_{h+1}\\
\mathsf{NEWROOT} &= \mathsf{ROOT}\,\langle p_0, \langle -\infty, k_{m+1}, \infty\rangle, \langle p, p', nil\rangle, put\rangle,
\end{aligned}$$

and if $k_{m+1} \leqslant k_h < k < k_{h+1}$, then

$$\begin{aligned}
N_1 &= \mathsf{NODE}\,\langle p, k_1 \cdots k_{m+1}, p_1 \cdots p_m\, p'\rangle\\
N_2 &= \mathsf{NODE}\,\langle p', k_{m+1} \cdots k_h\, k\, k_{h+1} \cdots k_{2m+1}, p_{m+1} \cdots p_h\, q\, p_{h+1} \cdots p_{2m+1}\rangle,
\end{aligned}$$

and if $k_h < k < k_{h+1} \leqslant k_{m+1}$ then

$$\begin{aligned}
N_1 &= \mathsf{NODE}\,\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_{m+1}, p_1 \cdots p_h\, q\, p_{h+1} \cdots p_m\, p'\rangle\\
N_2 &= \mathsf{NODE}\,\langle p', k_{m+1} \cdots k_{2m+1}, p_{m+1} \cdots p_{2m+1}\rangle.
\end{aligned}$$

$$\begin{aligned}
&\mathsf{STORE}(\tilde{p}, get, put, next) \stackrel{\text{df}}{=} \overline{get}(p_j).S +\\
&\qquad\qquad\qquad\qquad put(p).S' +\\
&\qquad\qquad\qquad\qquad next(p, n).\mathsf{cond}(p = p_h \rhd \bar{n}(p_{h+1}).S),
\end{aligned}$$

where $S = \mathsf{STORE}\,\langle \tilde{p}, get, put, next\rangle$ and $S' = \mathsf{STORE}\,\langle \tilde{p}p, get, put, next\rangle$.

$$\begin{aligned}
S_0(s, get) &\stackrel{\text{df}}{=} !\, s(k, a_s).get(p).S\,\langle k, p, a_s\rangle\\
S(k, p, a_s) &\stackrel{\text{df}}{=} (vr)\,\bar{p}(search_\langle k, r\rangle).r(y).\mathsf{case}(y)[link_p' : S\,\langle k, p', a_s\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad nonlink_p' : S\,\langle k, p', a_s\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad datum_b : \overline{a_s}(b).\mathbf{0}].
\end{aligned}$$

$$I_0(i, get, next)$$
$$\stackrel{\text{df}}{=} !\, i(k, b, a_i).get(p).Down\langle k, b, a_i, p, \langle\,\rangle\rangle$$
$$Down(k, b, a_i, p, \tilde{q})$$
$$\stackrel{\text{df}}{=} (vr)\ \bar{p}(search\_\langle k, r\rangle).r(y).$$
$$\mathsf{cond}(\tilde{q} = \langle\,\rangle \rhd \mathsf{case}(y)[link\_p' : Down\langle k, b, a_i, p', \langle p\rangle\rangle,$$
$$nonlink\_p' : Down\langle k, b, a_i, p', \langle p\rangle\rangle],$$
$$\tilde{q} \neq \langle\,\rangle \rhd \mathsf{case}(y)[link\_p' : Down\langle k, b, a_i, p', \tilde{q}\rangle,$$
$$nonlink\_p' : Down\langle k, b, a_i, p', p\tilde{q}\rangle,$$
$$datum\_b' : Insert\langle k, b, a_i, p, \tilde{q}\rangle])$$
$$Insert(k, b, a_i, p, q\tilde{q})$$
$$\stackrel{\text{df}}{=} (vr)\ \bar{p}(insert\_\langle k, b, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_p' : Insert\langle k, b, a_i, p', q\tilde{q}\rangle,$$
$$done\_\langle\,\rangle : \overline{a_i}.\mathbf{0},$$
$$split\_\langle p', k'\rangle : \overline{a_i}.Up\langle k', p', q, \tilde{q}\rangle]$$
$$Up(k, p, q, q_0\tilde{q})$$
$$\stackrel{\text{df}}{=} (vr)\ \bar{q}(add\_\langle k, p, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_q' : Up\langle k, p, q', q_0\tilde{q}\rangle,$$
$$done\_\langle\,\rangle : \mathbf{0},$$
$$split\_\langle p', k'\rangle : \mathsf{cond}(\tilde{q} \neq \langle\,\rangle \rhd Up\langle k', p', q_0, \tilde{q}\rangle,$$
$$\tilde{q} = \langle\,\rangle \rhd Up'\langle k', p', q_0, \langle q_0\rangle\rangle)]$$
$$Up'(k, p, q, \langle q_0\rangle)$$
$$\stackrel{\text{df}}{=} (vr)\ \bar{q}(add\_\langle k, p, r\rangle).r(y).$$
$$\mathsf{case}(y)[link\_q' : Up'\langle k, p, q', \langle q_0\rangle\rangle,$$
$$done\_\langle\,\rangle : \mathbf{0},$$
$$split\_\langle p', k'\rangle : (vn)\ \overline{next}(\langle q_0, n\rangle).n(q_1).Up'\langle k', p', q_1, \langle q_1\rangle\rangle].$$

$$P_0(s, i) \stackrel{\text{df}}{=} (vget, next)(S_0\langle s, get\rangle \mid I_0\langle i, get, next\rangle \mid T_0\langle get, next\rangle).$$

4. CORRECTNESS

We define an agent which gives a succinct description of the intended observable behaviour of the B^{link}-system P_0. This agent, B, is parametrized on a function f recording the key–pointer associations held in the leaves of the tree; a multiset σ (the *searches*) of pairs consisting of a key k to be searched for and a name a_s to be used to return the pointer found; a multiset σ^c (the *completed searches*) of pairs consisting of a name a_s and a pointer b found but not yet returned; a multiset ι (the *insertions*) of triples consisting of a key k, a pointer b, and a name a_i via which a signal is to be made when the insertion of the $\langle k, b\rangle$-pair has been completed; and a multiset ι^c (the *completed insertions*) of names a_i whose key–pointer pairs have been inserted but which have not been used to signal this.

We define B as follows. To ease readability, we elide the fixed parameters s and i from the instances of B on the right hand side of the definition.

$$\begin{aligned} B(s, i, f, \sigma, \sigma^c, \imath, \imath^c) \stackrel{\text{df}}{=} & s(k, a_s).B\langle f, \sigma \cup \{\langle k, a_s\rangle\}, \sigma^c, \imath, \imath^c\rangle \\ & + i(k, b, a_i).B\langle f, \sigma, \sigma^c, \imath \cup \{\langle k, b, a_i\rangle\}, \imath^c\rangle \\ & + \Sigma_{\langle k, a_s\rangle \in \sigma}\, \tau.B\langle f, \sigma - \{\langle k, a_s\rangle\}, \sigma^c \cup \{\langle a_s, f(k)\rangle\}, \imath, \imath^c\rangle \\ & + \Sigma_{\langle k, b, a_i\rangle \in \imath}\, \tau.B\langle f[b/k], \sigma, \sigma^c, \imath - \{\langle k, b, a_i\rangle\}, \imath^c \cup \{a_i\}\rangle \\ & + \Sigma_{\langle a_s, b\rangle \in \sigma^c}\, \overline{a_s}(b).B\langle f, \sigma, \sigma^c - \{\langle a_s, b\rangle\}, \imath, \imath^c\rangle \\ & + \Sigma_{a_i \in \imath^c}\, \overline{a_i}.B\langle f, \sigma, \sigma^c, \imath, \imath^c - \{a_i\}\rangle. \end{aligned}$$

The first and second summands represent initiation of new operations, the third and fourth invisible completion of outstanding operations (with appropriate update of the association in the case of insertion), and the fifth and sixth returns of results. Let $B_0 = B\langle i, s, \lambda k.nil, \varepsilon, \varepsilon, \varepsilon, \varepsilon\rangle$ where ε is the empty multiset.

When P_0 or B_0 receives a request to carry out an operation, the request contains a name via which to return the result. The claim

$$P_0 \simeq B_0$$

asserts that P_0 and B_0 are indistinguishable even allowing the possibility that different requests may contain the same return name. This claim is in fact correct and can be proved using a more complicated version of the theory presented below. We will assume, however, that each request for an operation contains a return name different from those in all other (active) requests. This natural assumption corresponds, for instance, to the use of integers in [8] to distinguish operation instances.

To express this assumption formally we introduce some notation. Working in the context given by Table 1, let A be the set of all names of types $\updownarrow \mathsf{D}$ and $\updownarrow\langle\ \rangle$. We refer to names of these types as A-names. If v is value we write v^{A} for the set of A-names occurring in v. Then we define $\simeq_{\mathsf{A}}$ to be the largest symmetric relation on agent such that

if $P \simeq_{\mathsf{A}} Q$ then for all actions α such that $\alpha \neq xv$ where $v^{\mathsf{A}} \cap \text{fn}(P, Q) \neq \varnothing$ and $\text{bn}(\alpha) \cap \text{fn}(P, Q) = \varnothing$, if $P \xrightarrow{\alpha} P'$ then (1) for some Q'', Q', $Q \Rightarrow Q'' \xrightarrow{\alpha} Q'$, $P \simeq_{\mathsf{A}} Q''$ and $P' \simeq_{\mathsf{A}} Q'$, or (2) $\alpha = \tau$ and $P' \simeq_{\mathsf{A}} Q$.

Thus, roughly, $P \simeq_{\mathsf{A}} Q$ if P and Q are branching bisimilar except that no requirement is placed on their behaviours on receiving an A-name that is free in one of them.

The assertion of correctness of the operations is then the following.

THEOREM 4.1. $P_0 \simeq_{\mathsf{A}} B_0$.

We give first a rough outline of the proof and then the proof itself. A central theme of the proof is the effective use of the theory of partial confluence. Roughly,

an agent is partial confluent if the occurrence of certain actions cannot preclude some others. A key insight is that in reasoning about the behaviour of a system constructed from partial-confluent parts, it is often sufficient to examine in detail only a part of that behaviour: from this and the fact of the parts' partial confluence, it may be possible to infer properties of the remaining behaviour of the system.

A derivative of P_0 is a restricted composition whose components are the replicators S_0 and I_0, instances of these representing operations in progress, and a derivative T of T_0 representing the data structure. In turn, T is a restricted composition whose components are a store and agents representing the nodes.

An interaction between an operation-agent and a node-agent takes the form of a *dialogue* consisting of a *question* via a name of type $\updownarrow \mathsf{P}$ (the node's name) and an *answer* via a name of type $\updownarrow \mathsf{R}$ supplied in the question. Let us call an operation-agent or node-agent *active* if it is engaged in a dialogue (i.e., it is a party to a question that has been asked but not answered) and *quiescent* if it is not. Let Q_0 be the part of P_0 whose states have at most one operation agent that is active in the sense just described. (This will be defined formally later.) Using the theory of partial confluence we will show that

$$P_0 \simeq Q_0. \tag{1}$$

We will further show that

$$Q_0 \simeq_{\mathsf{A}} B_0. \tag{2}$$

Since $P \simeq Q$ implies $P \simeq_{\mathsf{A}} Q$, the theorem follows immediately from (1) and (2).

To prove (2) we will extend the theory of partial confluence needed to establish (1). Let $Q_0^\flat$ be the part of Q_0 whose states have at most one operation-agent. Let $B_0^\flat$ also be the part of B_0 whose states are such that between them σ, σ^c, ι, and ι^c contain at most one element. $Q_0^\flat$ and $B_0^\flat$ can be thought of as the serial versions of the systems represented by Q_0 and B_0, respectively. We will show that as one would expect of these sequential parts,

$$Q_0^\flat \simeq B_0^\flat. \tag{3}$$

In addition we will show, using the partial confluence theory, that in each agent-context $\mathscr{C}[\,\cdot\,]$ in a certain class, Q_0 and the part $Q_0^\flat$ of it are indistinguishable, that is,

$$\mathscr{C}[\,Q_0^\flat\,] \simeq \mathscr{C}[\,Q_0\,], \tag{4}$$

and similarly that B_0 and the part $B_0^\flat$ of it are indistinguishable, that is,

$$\mathscr{C}[\,B_0^\flat\,] \simeq \mathscr{C}[\,B_0\,]. \tag{5}$$

From (3), (4), and (5) it follows that in each context $\mathscr{C}[\,\cdot\,]$ in the class, Q_0 and B_0 are indistinguishable, that is,

$$\mathscr{C}[Q_0] \simeq \mathscr{C}[B_0]. \tag{6}$$

Finally we will show by considering a particular context in the class that (2) follows from (6). The result (6) is important as the class of contexts contains all those in which one would expect to find the B^{link}-system.

We will point out that the analysis carried out to prove (4) can be extended to give an alternative proof of the theorem that does not use the partial confluence theory. Nonetheless, we believe that the theory of partial confluence is likely to be useful in other contexts and that the proof using it gives valuable insights into why the operations are correct. Further, it was as a result of carrying out the proof that we discovered the compression algorithm discussed in the next section.

We now proceed to the proof. In the next section we present the theory of partial confluence. In Section 4.2 we apply the theory to prove (1), (4), and (5), then show (3), and finally deduce (2). To finish the section we briefly discuss alternative models.

4.1. Partial Confluence

The theory of partial confluence can be presented in greater or lesser generality. We will not give here the most general development as it is not needed for the analysis and its complication may obscure some of the ideas. The following notation is useful to state an assumption made for the development here.

Notation 4.2. Let $\Delta = \{\updownarrow \sigma_1, ..., \updownarrow \sigma_n\}$ be a set of types. Given a context Γ, we write $x : \Delta$ and say x is a *Δ-name* if $\Gamma \vdash x : \updownarrow \sigma_i$ for some i. We write Δ^+ for the set of actions of the form xv with $x : \Delta$, and Δ^- for the set of actions of the form $\bar{x}(v\tilde{x})\,v$ with $x : \Delta$. We write $\Delta^{\pm}$ for $\Delta^+ \cup \Delta^-$ and refer to actions in $\Delta^{\pm}$ as *Δ-actions*. For a value v we set $v^{\Delta} = \{x : \Delta \mid x \text{ occurs in } v\}$.

Assumption 4.3. In the development of the theory of partial confluence that follows, we will assume, a context Γ and disjoint sets Q and A of types of the form $\updownarrow \sigma$ such that if $\Gamma \vdash \bar{x}v.P$ and $v^{\mathsf{A}} \neq \varnothing$, then $x : \mathsf{Q}$ and v^{A} contains just one occurrence of an A-name.

We will apply the theory in two cases. The first concerns interaction between operation-agents and node-agents. In that case Q will be $\{\updownarrow \mathsf{P}\}$ and A will be $\{\updownarrow \mathsf{R}\}$; an operation-agent interrogates a node-agent via an action $\bar{p}(vr)\,v$, where $p : \mathsf{Q}$ and $v^{\mathsf{A}} = \{r\}$. The second case concerns interaction between the data structure and its environment. In that case Q will be $\{\updownarrow\langle \mathsf{int}, \updownarrow \mathsf{D}\rangle, \updownarrow\langle \mathsf{int}, \mathsf{D}, \updownarrow\langle\,\rangle\rangle\}$ and A will be $\{\updownarrow \mathsf{D}, \updownarrow\langle\,\rangle\}$; the environment requests a search operation via an action $\bar{s}(va_s)\,v$, where $s : \mathsf{Q}$ and $v^{\mathsf{A}} = \{a_s\}$. In general we will think of a Q-action as the asking of a question in which the A-name to be used to return the answer is passed from questioner to answerer.

We first recall the basic theory of partial confluence from [7] to which we refer for proofs. (In fact the presentation here gives a more general treatment of bound names, but the proofs are very similar.) We begin with some useful notation.

Notation 4.4. $P \Rightarrow \xrightarrow{\alpha} P'$ means that $P \Rightarrow P'' \xrightarrow{\alpha} P'$ for some P'' with $P'' \simeq P$ and if $\alpha = \tau$ then $P' \not\simeq P$. (This is a nonstandard use of notation, but confusion is unlikely to result because the use is followed throughout the paper.) For $s = \alpha_1 \cdots \alpha_n \in Act^*$ we write $\xrightarrow{s}_{\simeq}$ for the composite relation $\simeq \xrightarrow{\alpha_1} \simeq \cdots \simeq \xrightarrow{\alpha_n} \simeq$, where in writing $P \simeq \xrightarrow{\tau} \simeq P'$ we intend $P' \not\simeq P$. We refer to $\xrightarrow{\tau}_{\simeq}$ as a *decisive* (silent) transition or action.

The essence of partial confluence is that the occurrence of some actions, here the A-actions, will never preclude others and that behaviour on A-actions is determinate. A subtle point is the treatment of bound names in output actions. Consider $P = (\nu p)(\bar{a}(p).\mathbf{0} \mid \bar{b}(p).\mathbf{0})$ where a, b are A-names. Then $P \xrightarrow{\alpha} P_b = \bar{b}(p).\mathbf{0}$, where $\alpha = \bar{a}(\nu p)\, p$, and $P \xrightarrow{\beta} P_a = \bar{a}(p).\mathbf{0}$, where $\beta = \bar{b}(\nu p)\, p$. Then $P_b \xrightarrow{\bar{p}b} \mathbf{0}$ rather than $P_b \xrightarrow{\beta} \mathbf{0}$. We do not, however, wish to regard $P \xrightarrow{\alpha} P_b$ as precluding $P \xrightarrow{\beta} P_a$, although it does affect which names are bound in the action. To capture this we introduce some notation. Given a pair of actions α, β, we write α^*, β^* for α, β unless $\alpha = \bar{a}(\nu \tilde{x})\, v$ and $\beta = \bar{b}(\nu \tilde{y})\, u$ when α^* is $\bar{a}(\nu \tilde{x} - \tilde{y})\, v$ and β^* is $\bar{b}(\nu \tilde{y} - \tilde{x})\, u$. (We assume that bound names are chosen so that no name occurs bound more than once in an agent and that the bound names in an action of an agent are the corresponding bound names in the agent.) Then

DEFINITION 4.5. 1. An agent P is *weak* A-*confluent* if, whenever Q is a derivative of P, $\alpha \in \mathsf{A}^{\pm}$, $Q \xrightarrow{\alpha} Q_1$, and $Q \Rightarrow \xrightarrow{\beta} Q_2$, then $Q_1 \Rightarrow \xrightarrow{\beta^*} Q'$ and $Q_2 \Rightarrow \xrightarrow{\alpha^*} \simeq Q'$, or $\alpha = \beta$ and $Q_1 \simeq Q_2$.

2. An agent P is A-*determinate* if whenever Q is a derivative of P,

(a) if $\alpha \in \mathsf{A}^+$, $Q \xrightarrow{\alpha} Q_1$, and $Q \Rightarrow \xrightarrow{\alpha} Q_2$, then $Q_1 \simeq Q_2$, and

(b) if α, $\beta \in \mathsf{A}^-$ with $\mathsf{subj}(\alpha) = \mathsf{subj}(\beta)$, $Q \xrightarrow{\alpha} Q_1$, and $Q \Rightarrow \xrightarrow{\beta} Q_2$, then $\alpha = \beta$ and $Q_1 \simeq Q_2$.

3. An agent P is A-*confluent* if it is weak A-confluent and A-determinate.

A derivative of a weak A-confluent agent enjoys a confluence property with respect to A-actions and other actions. A derivative of an A-determinate agent is determinate under A-actions. Note that a weak A-confluent agent need not be semantically invariant under τ-transitions. It is straightforward to show that if P is weak A-confluent and $Q \simeq P$, then Q is weak A-confluent, and similarly for A-determinacy; see [7].

We record an observation that will be useful later. It describes how actions of a weak A-confluent agent commute with sequences of $\mathsf{A}^{\pm}$-actions. The appropriate form of commuting involves the "*"-notation introduced above extended in the natural way to sequences.

LEMMA 4.6. *Suppose* P *is weak* A-*confluent*, $P \xrightarrow{s}_{\simeq} P_1$, *where* $s \in \mathsf{A}^{\pm}{}^*$, *and* $P \Rightarrow \xrightarrow{\beta} P_2$. *Then*

1. $P \Rightarrow \xrightarrow{\beta^*} P_3$ *and* $P_2 \xrightarrow{s^*}_{\simeq} P_3$, *or*

2. $s = t\alpha u$, $P \xrightarrow{t}_{\simeq} P_0 \xrightarrow{\alpha}_{\simeq} P_3 \xrightarrow{u}_{\simeq} P_1$, $P_0 \Rightarrow \xrightarrow{\beta^*} P_4$, $P_2 \xrightarrow{t^*}_{\simeq} P_4$, $\alpha = \beta^*$, *and* $P_4 \simeq P_3$.

Proof. The proof is by induction on the length of s and is straightforward from the definitions. ∎

A further convenient definition:

DEFINITION 4.7. An agent is A-*closed* if none of its derivatives can perform an A-action.

The following important result asserts that $\simeq$-state (that is, the equivalence class under $\simeq$) of an A-closed composition of A-confluent agents is not altered by an interaction between components via an A-name. We say that actions α, $\bar{\alpha}$ are *complementary*, α comp $\bar{\alpha}$, if $\alpha = xv$ and $\bar{\alpha} = \bar{x}(v\tilde{x})\, v$ or vice versa.

LEMMA 4.8. *Suppose C and S are* A-*confluent*, $(v\tilde{z})(C \mid S)$ *is* A-*closed, and* $C \xrightarrow{\alpha}_{\simeq} C'$, $S \xrightarrow{\bar{\alpha}}_{\simeq} S'$, *where* α comp $\bar{\alpha}$ *and* $\bar{\alpha} \in \mathsf{A}^-$. *Then* $(v\tilde{z})(C \mid S) \simeq (v\tilde{z}\tilde{u})(C' \mid S')$, *where* $\tilde{u} = \mathrm{bn}(\bar{\alpha})$.

Proof. See [7.] ∎

Note that this does not hold under the assumption only that C, S are weak A-confluent: consider $C_1 = \bar{a}(b).\mathbf{0} \mid \bar{a}(c).\mathbf{0}$ and $S_1 = a(x).a(y).\bar{x}.\mathbf{0}$, and $C_2 = \bar{a}(b).\bar{a}(c).\mathbf{0}$ and $S_2 = a(x).\bar{x}.\mathbf{0} \mid a(y).\mathbf{0}$ where C_1 and S_2 are not A-determinate.

The following definition gives the part of an agent's behaviour in which it never receives an A-name that it already has and never sends a free A-name.

DEFINITION 4.9. Let $\mathscr{P}$ be the transition system generated by an agent P. The subsystem $\langle \mathscr{P}_\natural, \{\xrightarrow{\alpha}_\natural\}_\alpha \rangle$ of $\mathscr{P}$ is the smallest transition system such that

1. $P \in \mathscr{P}_\natural$, and
2. if $Q \in \mathscr{P}_\natural$, and $Q \xrightarrow{\alpha} R$, and neither (i) $\alpha = xv$, where $v^{\mathsf{A}} \cap \mathrm{fn}(Q) \neq \varnothing$ nor (ii) $\alpha = \bar{x}(v\tilde{x})\, v$, where $v^{\mathsf{A}} \not\subseteq \tilde{x}$, then $R \in \mathscr{P}_\natural$ and $Q \xrightarrow{\alpha}_\natural R$.

Thus $\langle \mathscr{P}_\natural, \{\xrightarrow{\alpha}_\natural\}_\alpha \rangle$ consists of some of the derivatives of P and some of the transitions between those agents. The transitions excluded are those in which an agent receives an A-name that occurs free in it or sends a free A-name, and the derivatives excluded are those reachable from P only via computations involving excluded actions. When considering a derivative Q of P that is in $\mathscr{P}_\natural$, we often write $Q^\natural$ instead of Q to emphasise the transition system that is under consideration.

Hereafter, we write $\mathrm{fn}(Q)^{\Delta}$ for $\{x \in \mathrm{fn}(Q) \mid x : \Delta\}$. We continue with a final piece of notation:

Notation 4.10. For $\tilde{a}$ a set of A-names let $\mathscr{P}^{\tilde{a}}_\natural = \{Q \in \mathscr{P}_\natural \mid \mathrm{fn}(Q)^{\mathsf{A}} = \tilde{a}\}$. Note that the sets $\mathscr{P}^{\tilde{a}}_\natural$ partition $\mathscr{P}_\natural$.

The following definition isolates a class of questioners.

DEFINITION 4.11. An agent P is an A-*client* if $P^\natural$ is A-confluent and

1. whenever $Q \in \mathscr{P}^{\tilde{a}}_\natural$ and $Q \xrightarrow{\alpha}_\natural R$, then
   (a) if $\alpha = \bar{x}(v\tilde{x})\, v$ with $v^{\mathsf{A}} = \{a\}$, then $R \in \mathscr{P}^{\tilde{a}a}_\natural$,
   (b) if $\alpha = av$ with a an A-name, then $R \in \mathscr{P}^{\tilde{a}-a}_\natural$,
   (c) if α is of some other form then $R \in \mathscr{P}^{\tilde{a}}_\natural$, and
2. whenever $Q \in \mathscr{P}^{\tilde{a}}_\natural$, $a \in \tilde{a}$, and $\alpha \in \mathsf{A}^+$ with $\mathsf{subj}(\alpha) = a$, then $Q \xrightarrow{\alpha}_\natural$.

It is intended that if $Q \in \mathscr{P}_\natural^{\tilde{a}}$ then the index set $\tilde{a}$ contains the names on which the A-client expects an answer. When a question is asked a fresh A-name is sent and added to the index set (1a). When an answer is received the name used is deleted from the index set (1b). Other actions do not change the index set (1c). Finally, an A-client is ready to receive an answer via any name in the index set (2).

Complementing this is the following definition of a class of answerers.

DEFINITION 4.12. An agent P is an A-*server* if $P^\natural$ is A-confluent and whenever $Q \in \mathscr{P}_\natural^{\tilde{a}}$ and $Q \xrightarrow{\alpha}_\natural R$, then

1. if $\alpha = xv$ with $v^{\mathsf{A}} = \{a\}$, then $R \in \mathscr{P}_\natural^{\tilde{a}a}$,
2. if $\alpha = \bar{a}(v\tilde{x})\, v$ with a an A-name, then $R \in \mathscr{P}_\natural^{\tilde{a}-a}$, and
3. if α is of some other form then $R \in \mathscr{P}_\natural^{\tilde{a}}$.

If $Q \in \mathscr{P}_\natural^{\tilde{a}}$ then $\tilde{a}$ contains the names on which the A-server owes an answer. When a question is received the A-name (which is fresh by definition of $\rightarrow_\natural$) is added to the index set (1), and when a question is answered the name used is deleted from the set (2). Other actions do not change the index set (3).

In outlining the structure of the proof of Theorem 4.1, we referred to parts of agents whose derivatives have at most one component of a certain kind. The following definition makes this precise.

DEFINITION 4.13. Let $\mathscr{P}$ be the transition system generated by an agent P with $\mathrm{fn}(P)^{\mathsf{A}} = \varnothing$. Then $\langle \mathscr{P}_\flat, \{\xrightarrow{\alpha}_\flat\}_\alpha \rangle$ is the subsystem of $\langle \mathscr{P}_\natural, \{\xrightarrow{\alpha}_\natural\}_\alpha \rangle$ with $\mathscr{P}_\flat = \bigcup \{\mathscr{P}_\natural^{\tilde{a}} \mid |\tilde{a}| \leqslant 1\}$ and $Q \xrightarrow{\alpha}_\flat R$ if $Q, R \in \mathscr{P}_\flat$, and $Q \xrightarrow{\alpha}_\natural R$.

Thus $\langle \mathscr{P}_\flat, \{\xrightarrow{\alpha}_\flat\}_\alpha \rangle$ consists of the derivatives of P that have at most one free A-name and the transitions between them (that respect that A-name property). When considering a derivative Q of P that is in $\mathscr{P}_\flat$, we often write $Q^\flat$ instead of Q to emphasise the transition system that is under consideration.

In what follows we will write an expression such as $(v\tilde{z})(R \mid P^\flat)$. Such an expression describes the part of the transition system of $(v\tilde{z})(R \mid P)$ containing the agents of the form $(v\tilde{w})(R' \mid Q)$ where Q is in $\mathscr{P}_\flat$ and whose transitions are the transitions between these agents. (Note that every derivative of $(v\tilde{z})(R \mid P)$ is of the form $(v\tilde{w})(R' \mid P')$ where R' is a derivative of R and P' is a derivative of P.)

The following definition picks out a class of A-servers.

DEFINITION 4.14. An A-server P is *prompt* if whenever $Q \in \mathscr{P}_\natural^a$ with a a singleton, then $Q \Rightarrow \xrightarrow{\alpha}$ for some $\alpha = \bar{a}(v\tilde{x})\, v$.

If $Q \in \mathscr{P}_\natural^a$ then Q may deliver the answer to the one outstanding question, possibly after some τ-actions that do not change its $\simeq$-state.

We now have the first main result. It asserts that a prompt A-server S is indistinguishable from the part $S^\flat$ of it in an A-closed composition with an A-client.

THEOREM 4.15. *Suppose C is an* A-*client, S is a prompt* A-*server,* $\mathrm{fn}(C)^{\mathsf{A}} = \varnothing$, $\mathrm{fn}(S)^{\mathsf{A}} = \varnothing$, *and* $(v\tilde{z})(C \mid S)$ *is* A-*closed. Then* $(v\tilde{z})(C \mid S) \simeq (v\tilde{z})(C \mid S^\flat)$.

Proof. See [7]. ∎

The definitions and results presented so far in this section are generalizations of material in [7] and constitute a basic theory of partial confluence which will be used in the next section. We continue with an extension of the theory.

In Theorem 4.15, S is assumed to be prompt: on accepting a question it immediately assumes a state in which it may deliver the answer, possibly after some semantically-insignificant τ-actions. All nodes of the B^{link} data-structure are prompt. The B^{link}-system, however, is not prompt (for the appropriate Q and A): determination of the result of an operation may involve a state change affecting subsequent operations. The extension of the theory is concerned with servers in which determination of the answer to a question may involve at most one decisive τ-action.

It involves agents some of whose actions commute, where the term is understood to take into account possible change of bound names. Consider again $P=(vp)(\bar{a}(p).\mathbf{0} \mid \bar{b}(p).\mathbf{0})$. Then $P \xrightarrow{\alpha} \xrightarrow{\beta} \mathbf{0}$ where $\alpha=\bar{a}(vp)\ p$ and $\beta=\bar{b}p$. The communication is $P \xrightarrow{\beta^\dagger} \xrightarrow{\alpha^\dagger} \mathbf{0}$, where $\alpha^\dagger=\bar{a}p$ and $\beta^\dagger=\bar{b}(vp)\ p$. In the following definitions, given a pair of actions α, β we write $\alpha^\dagger$, $\beta^\dagger$ for α, β unless $\alpha=\bar{a}(v\tilde{x})\ v$ and $\beta=\bar{b}(v\tilde{y})\ u$ when for some $\tilde{z}\subseteq\tilde{x}$, $\alpha^\dagger$ is $\bar{a}(v\tilde{x}-\tilde{z})\ v$ and $\beta^\dagger$ is $\bar{b}(v\tilde{y}\tilde{z})\ u$. Also, we say actions α, β are *partners* if either $\alpha=\bar{x}(v\tilde{x})\ v$ and $\beta=au$, where $v^{\mathsf{A}}=\{a\}$, or $\alpha=xv$ and $\beta=\bar{a}(v\tilde{y})\ u$, where $v^{\mathsf{A}}=\{a\}$.

DEFINITION 4.16. An A-client P is a QA-*client* if $P^\natural$ is weak Q-confluent and for $Q\in\mathscr{P}_\natural$, if $Q\Rightarrow \xrightarrow{\alpha} Q_1 \Rightarrow \xrightarrow{\beta} Q_2$, $\alpha\in\mathsf{Q}^-$, and α, β are not partners, then $Q\Rightarrow \xrightarrow{\beta^\dagger} Q_1' \Rightarrow \xrightarrow{\alpha^\dagger} Q_2'$ with $Q_2\simeq Q_2'$.

A QA-client is weak Q-confluent as well as A-confluent, and the asking of a question commutes with any action except the receipt of the answer to that question.

We now isolate a class of servers motivated by consideration of the B^{link}-system.

DEFINITION 4.17. An A-server P is an *almost-prompt* QA-*server* if $P^\natural$ is Q-confluent and for $Q\in\mathscr{P}_\natural$,

1. if $Q\Rightarrow \xrightarrow{\alpha} Q_1 \Rightarrow \xrightarrow{\beta} Q_2$ and (a) $\alpha\in\mathsf{Q}^+$, α, β are not partners, and $\beta\neq\tau$, or (b) $\beta\in\mathsf{A}^-$, α, β are not partners, and $\alpha\neq\tau$, or (c) $\beta=\tau$ and $\alpha\notin(\mathsf{Q}^+\cup\mathsf{A}^-\cup\{\tau\})$, then $Q\Rightarrow \xrightarrow{\beta^\dagger} Q_1' \Rightarrow \xrightarrow{\alpha^\dagger} Q_2'$ with $Q_2\simeq Q_2'$,

2. if $Q\in\mathscr{P}_\natural^a$ (where a is a singleton) then $Q\Rightarrow \xrightarrow{\alpha}$ or $Q\Rightarrow \xrightarrow{\tau} Q' \Rightarrow \xrightarrow{\alpha}$ for some $\alpha=\bar{a}(v\tilde{x})\ v$,

3. if $Q\Rightarrow \xrightarrow{\tau} Q'$ then there exists $\alpha\in\mathsf{A}^-$ such that $Q'\Rightarrow \xrightarrow{\alpha}$ but not $(Q\Rightarrow \xrightarrow{\alpha'})$ for any α' with $\mathsf{subj}(\alpha')=\mathsf{subj}(\alpha)$,

4. if not $(Q\Rightarrow \xrightarrow{\alpha})$, $Q\Rightarrow \xrightarrow{\tau} Q_1\Rightarrow \xrightarrow{\alpha}$, and $Q\Rightarrow \xrightarrow{\tau} Q_2\Rightarrow \xrightarrow{\alpha'}$, where α, $\alpha'\in\mathsf{A}^-$ with $\mathsf{subj}(\alpha')=\mathsf{subj}(\alpha)$, then $Q_1\simeq Q_2$.

Note that a QA-server is Q-determinate. Condition 1 asserts that certain actions commute. The second condition expresses that when there is one outstanding question it may be answered after at most one decisive τ-action. Condition 3 expresses that each decisive τ-action determines the answer to a question and Condition 4 that only one decisive τ-action can determine the answer to a particular question.

The following result records some properties of almost-prompt QA-servers. For $J \subseteq Act$, an agent P is *J-inert* if for all $\alpha \in J$, not $(P \Rightarrow \xrightarrow{\alpha})$.

LEMMA 4.18. *Suppose P is an almost-prompt* QA*-server.*

1. *If $Q \in \mathscr{P}^{\varnothing}_{\natural}$ then Q is $\mathsf{A}^- \cup \{\tau\}$-inert.*
2. *If $Q \in \mathscr{P}^{a}_{\natural}$ is not A^--inert then Q is τ-inert.*
3. *If $Q \in \mathscr{P}^{\varnothing}_{\natural}$, $Q \xrightarrow{u}_{\simeq} Q_1 \xrightarrow{\beta}_{\simeq} \xrightarrow{\alpha}_{\simeq} Q_2$, where $u\beta \in \mathsf{Q}^{+}*$ and $\alpha \in \mathsf{A}^-$ and Q_1 is A^--inert, then Q_2 in A^--inert.*

Proof. The first part is straightforward from the definitions.

For the second suppose that $Q \Rightarrow \xrightarrow{\alpha} Q_1$, where $\alpha \in \mathsf{A}^-$. If $Q \Rightarrow \xrightarrow{\tau}$ then as Q is QA-confluent, $Q_1 \Rightarrow \xrightarrow{\tau}$, contrary to the first part as $Q_1 \in \mathscr{P}^{\varnothing}_{\natural}$.

For the third part suppose $Q \in \mathscr{P}^{\varnothing}_{\natural}$, $Q \xrightarrow{u}_{\simeq} Q_1 \xrightarrow{\beta}_{\simeq} \xrightarrow{\alpha}_{\simeq} Q_2$, where $u\beta \in \mathsf{Q}^{+}*$, $\alpha \in \mathsf{A}^-$, and Q_1 is A^--inert. If Q_2 is not A^--inert then $Q_2 \Rightarrow \xrightarrow{\alpha'}$, where $\alpha' \in \mathsf{A}^-$, when either $Q_1 \xrightarrow{\alpha}_{\simeq}$ or $Q_1 \xrightarrow{\alpha'}_{\simeq}$ since at least one of α, α' commutes with β and α' commutes with α. But this contradicts that Q_1 is A^--inert. ∎

The main result, Theorem 4.20 below, asserts that an almost-prompt QA-server S is indistinguishable from the part $S^{\flat}$ of it in a QA-closed composition with a QA-client. The following result is used in its proof. In asserts that the $\simeq$-state of a QA-closed composition of a QA-client and a pruned almost-prompt QA-server is not changed by a communication via an A-name.

THEOREM 4.19. *Suppose that C is a* QA*-client with state space $\mathscr{C}$ and S is an almost-prompt* QA*-server with state space $\mathscr{S}$. Suppose $C_1 \in \mathscr{C}^{a}_{\natural}$, $S_1 \in \mathscr{S}^{a}_{\natural}$, and $C_1 \xrightarrow{\alpha} C_2$ and $S_1 \overset{\bar{\alpha}}{\Rightarrow} S_2$ where $\bar{\alpha} \in \mathsf{A}^-$ and α comp $\bar{\alpha}$. Further suppose $(\nu\tilde{z})(C_1 \mid S_1)$ is* QA*-closed. Then $(\nu\tilde{z})(C_1 \mid S_1^{\flat}) \simeq (\nu\tilde{z}\tilde{u})(C_2 \mid S_2^{\flat})$, where $\tilde{u} = \mathrm{bn}(\bar{\alpha})$.*

Proof. Since $S_1 \in \mathscr{S}^{a}_{\natural}$, either $S_1(\Rightarrow \xrightarrow{\bar{\alpha}}) \Rightarrow S_2$ or $S_1(\Rightarrow \xrightarrow{\tau})(\Rightarrow \xrightarrow{\bar{\alpha}}) \Rightarrow S_2$. Let $(M_1, M_2) \in \mathscr{B}_0$ if $M_1 = (\nu\tilde{z})(C_1 \mid S_1^{\flat})$ and $M_2 = (\nu\tilde{z}\tilde{u})(C_2 \mid S_2^{\flat})$, where $C_1 \in \mathscr{C}^{a}_{\natural}$, $S_1 \in \mathscr{S}^{a}_{\natural}$, $C_1 \xrightarrow{\alpha}_{\simeq} C_2$ $S_1 \xrightarrow{\bar{\alpha}}_{\simeq} S_2$, and $\alpha = av$, $\bar{\alpha} = \bar{a}(v\tilde{u})\, v$. Moreover, let $(M_1, M_2) \in \mathscr{B}_1$ if $M_1 = (\nu\tilde{z})(C_1 \mid S_1^{\flat})$ and $M_2 = (\nu\tilde{z}\tilde{u})(C_2 \mid S_2^{\flat})$, where $C_1 \in \mathscr{C}^{a}_{\natural}$, $S_1 \in \mathscr{S}^{a}_{\natural}$, $C_1 \xrightarrow{\alpha}_{\simeq} C_2$, $S_1 \xrightarrow{\tau}_{\simeq} S_0 \xrightarrow{\bar{\alpha}}_{\simeq} S_2$, $\alpha = av$ and $\bar{\alpha} = \bar{a}(v\tilde{u})\, v$. We show that $\mathscr{B}_0 \cup \mathscr{B}_1 \cup \simeq$ is a branching bisimulation.

First suppose $(M_1, M_2) \in \mathscr{B}_0$, where M_1, M_2 are as above. Suppose $M_2 \xrightarrow{\rho} M_2'$. Then $M_1 \Rightarrow M_1'' = (\nu\tilde{z}\tilde{u})(C_1' \mid S_1'^{\flat})$, where $C_1 \Rightarrow \xrightarrow{\alpha} C_1' \simeq C_2$ and $S_1 \Rightarrow \xrightarrow{\bar{\alpha}} S_1' \simeq S_2$. Since $S_1' \simeq S_2$, $S_1'^{\flat} \simeq S_2^{\flat}$. Hence $M_1'' \simeq M_2$ which implies that $M_1'' \Rightarrow \xrightarrow{\rho} M_1'$, where $M_1' \simeq M_2'$ as required. (The case $\rho = \tau$ and $M_2' \simeq M_2$ is clear.) So suppose $M_1 \xrightarrow{\rho} M_1'$. The following possibilities exist:

1. $M_1' \equiv (\nu\tilde{y})(C_1' \mid S_1^{\flat})$, where $C_1 \xrightarrow{\beta} C_1'$. Since C_1 is QA-confluent, $C_2 \Rightarrow \xrightarrow{\beta} C_2'$ and $C_1' \xrightarrow{\alpha}_{\simeq} C_2'$. Hence $M_2 \Rightarrow \xrightarrow{\beta} M_2' = (\nu\tilde{y}\tilde{u})(C_2' \mid S_2^{\flat})$, where $(M_1', M_2') \in \mathscr{B}_0$.

2. $M_1' \equiv (\nu\tilde{z})(C_1 \mid S_1'^{\flat})$, where $S_1 \xrightarrow{\tau} S_1'$. By Lemma 4.18(2), $S_1 \simeq S_1'$. So $S_1' \xrightarrow{\bar{\alpha}}_{\simeq} S_2$ and hence $(M_1', M_2) \in \mathscr{B}_0$.

3. $M_1' \equiv (\nu\tilde{y})(C_1 \mid S_1'^{\flat})$, where $S_1 \xrightarrow{\beta} S_1'$ and $\beta \neq \tau$. The argument is similar to the first above with attention to bound names.

4. $M_1' \equiv (\nu\tilde{z}\tilde{w})(C_1' \mid S_1'^{\flat})$, $C_1 \xrightarrow{\gamma} C_1'$, $S_1 \xrightarrow{\bar{\gamma}} S_1'$, where γ comp $\bar{\gamma}$, $\tilde{w} = \mathrm{bn}(\gamma, \bar{\gamma})$. Note that since $S_1 \in \mathscr{S}^a_{\natural}$, $\gamma \notin \mathsf{Q}^-$. If $\bar{\gamma} \in \mathsf{A}^-$ then $\mathsf{subj}(\bar{\gamma}) = \bar{a}$, and so by A-determinacy $\bar{\gamma} = \bar{\alpha}$ and $S_1' \simeq S_2$. Also, $\gamma = \alpha$ and by A-determinacy, $C_1' \simeq C_2$. Hence $M_1' \simeq M_2$. Otherwise the argument is a combination of those above.

So suppose $(M_1, M_2) \in \mathscr{B}_1$. As before it is clear that if $M_2 \xrightarrow{\rho} M_2'$ then $M_1 \Rightarrow M_1'' = (\nu\tilde{z}\tilde{u})(C_1' \mid S_1'^{\flat})$, where $C_1 \Rightarrow \xrightarrow{\alpha} C_1' \simeq C_2$ and $S_1 \Rightarrow \xrightarrow{\tau} \Rightarrow \xrightarrow{\bar{\alpha}} S_1' \simeq S_2$. Hence $M_1'' \simeq M_2$. Moreover, $M_1'' \Rightarrow \xrightarrow{\rho} M_1'$ where $M_1' \simeq M_2'$ as required. (Again the case $\rho = \tau$ and $M_2' \simeq M_2$ is clear.) So suppose $M_1 \xrightarrow{\rho} M_1'$. The following possibilities exist:

1. $M_1' \equiv (\nu\tilde{y})(C_1' \mid S_1^{\flat})$ and $C_1 \xrightarrow{\beta} C_1'$. Then $C_2 \Rightarrow \xrightarrow{\beta} C_2'$ and $C_1' \xrightarrow{\alpha}_{\simeq} C_2'$ so $M_2 \Rightarrow \xrightarrow{\beta} M_2' = (\nu\tilde{y}\tilde{u})(C_2' \mid S_2^{\flat})$, where $(M_1', M_2') \in \mathscr{B}_1$.

2. $M_1' \equiv (\nu\tilde{z})(C_1 \mid S_1'^{\flat})$, where $S_1 \xrightarrow{\tau} S_1'$. If $S_1 \simeq S_1'$ then $S_1' \xrightarrow{\tau}_{\simeq} \xrightarrow{\bar{\alpha}}_{\simeq} S_2$ and so $(M_1', M_2) \in \mathscr{B}_1$ as required. Otherwise, as S is almost-prompt, $S_1' \simeq S_0$ when $S_1' \xrightarrow{\bar{\alpha}}_{\simeq} S_2$. Hence $(M_1', M_2) \in \mathscr{B}_0$.

3. $M_1' \equiv (\nu\tilde{y})(C_1 \mid S_1'^{\flat})$, where $S_1 \xrightarrow{\beta} S_1'$ and $\beta \neq \tau$. Note that since $S_1 \xrightarrow{\tau}_{\simeq} S_0 \xrightarrow{\bar{\alpha}}_{\simeq} S_2$ and $S_1 \in \mathscr{S}^a_{\natural}$, by Lemma 4.18(2), S_1 is A^--inert. Since $S_1' \in \mathscr{S}^a_{\natural}$, either $S_1' \Rightarrow \xrightarrow{\gamma}$ or $S_1' \Rightarrow \xrightarrow{\tau} \Rightarrow \xrightarrow{\gamma} S_1''$, where $\gamma \in \mathsf{A}^-$, $\mathsf{subj}(\gamma) = a$. In the former case by commutativity, $S_1 \Rightarrow \xrightarrow{\gamma^\dagger}$ contrary to S_1 being A^--inert. (Note that $\beta \neq \tau$ by assumption, and β, γ are not partners as $\beta \notin \mathsf{Q}^-$ by Q-closure.) Hence by commutativity $S_1 \xrightarrow{\tau}_{\simeq} S_0' \xrightarrow{\gamma^\dagger}_{\simeq} S_2' \xrightarrow{\beta^\dagger}_{\simeq} S_1''$. So as S is an almost-prompt QA-sever, $S_0'' \simeq S_0$, $\gamma^\dagger = \bar{\alpha}$, and $S_2' \simeq S_2$ so $S_2 \Rightarrow \xrightarrow{\beta^\dagger} S_2' \simeq S_1'$. Hence $(M_1', (\nu\tilde{y}\tilde{u})(C_2 \mid S_2'^{\flat})) \in \mathscr{B}_0$.

4. $M_1' \equiv (\nu\tilde{z}\tilde{w})(C_1' \mid S_1'^{\flat})$, $C_1 \xrightarrow{\gamma} C_1'$, $S_1 \xrightarrow{\bar{\gamma}} S_1'$, γ comp $\bar{\gamma}$ and $\tilde{w} = \mathrm{bn}(\gamma, \bar{\gamma})$. Note that since $S_1 \in \mathscr{S}^a_{\natural}$, $\gamma \notin \mathsf{Q}^-$. Also $\bar{\gamma} \notin \mathsf{A}^-$ as S_1 as A^--inert. The argument is then a combination of those above.

This completes the proof. ▮

The intuition underlying the main theorem below is that the determination of the answer to any question can be thought of as an atomic action, and therefore processing of questions can be serialized. The reason for this is that the answer to a question is determined by at most one decisive τ-action, and each decisive τ-action determines an answer to a question. In more detail, let P be a system with a server capable of processing questions concurrently, and let P' be a similar system with a serial server. Clearly, any computation of P' is also (essentially) a computation of P. The converse, however, does not hold: there are states of P where more than one question is outstanding in the server. Nonetheless, P' is branching bisimilar to P. A key observation in seeing that P' can match any computation of P is that the server of P' can postpone accepting a question which has been accepted by the server of P until the decisive τ-action (if it exists) is performed. The serial system can then accept the question and determine and return the answer. Since the answer to the question has been determined in P, by A-confluence no difference in behaviour is observable. This intuition is formalized in the proof.

THEOREM 4.20. *Suppose that C is a* QA*-client and that S is an almost-prompt* QA*-server. Suppose* $\mathrm{fn}(C)^{\mathsf{A}} = \varnothing$, $\mathrm{fn}(S)^{\mathsf{A}} = \varnothing$, *and* $(\nu\tilde{z})(C \mid S)$ *is* QA*-closed. Then* $(\nu\tilde{z})(C \mid S) \simeq (\nu\tilde{z})(C \mid S^{\flat})$.

Proof. Let $(M_1, M_2) \in \mathscr{B}$ if $M_1 = (\nu\tilde{z})(C_1 \mid S_1)$ and $M_2 = (\nu\tilde{w})(C_2 \mid S_2^\flat)$, where $C_2 \in \mathscr{C}_\natural^\varnothing$, $S_2 \in \mathscr{S}_\natural^\varnothing$ and there are C, an A^--inert S, $\bar{s} = \overline{\alpha_1} \cdots \overline{\alpha_n} \in \mathsf{A}^-$ with $\overline{\alpha_i} = \overline{a_i}(\nu\tilde{x}_i)\, v_i$, and $\bar{u} = \overline{\beta_1} \cdots \overline{\beta_k} \in \mathsf{Q}^-$ with $\overline{\beta_i} = \overline{b_i}(\nu\tilde{y}_i)\, u_i$ such that setting $s = \alpha_1 \cdots \alpha_n$, where $\alpha_i = a_i v_i$ and $u = \beta_1 \cdots \beta_k$, where $\beta_i = b_i u_i$, then $C_1 \xrightarrow{s} \simeq C$ and $S_1 \xrightarrow{\bar{s}} \simeq S$, and $C_2 \xrightarrow{\bar{u}} \simeq C$ and $S_2 \xrightarrow{u} \simeq S$. We show that $\mathscr{B} \simeq \cup \simeq$ is a branching bisimulation.

First suppose $(M_1, M_2) \in \mathscr{B}$, where M_1, M_2 are as above. Suppose $M_2 \xrightarrow{\rho} M_2'$. We show that $M_1 \Rightarrow M_1'' \xrightarrow{\rho} M_1'$, where $(M_1'', M_2) \in \mathscr{B} \simeq$ and $(M_1', M_2') \in \mathscr{B} \simeq$ (or $\rho = \tau$ and $M_2' \simeq M_2$ when $(M_1, M_2') \in \mathscr{B} \simeq$). There are several cases.

1. $M_2' \equiv (\nu\tilde{y})(C_2' \mid S_2^\flat)$, where $C_2 \xrightarrow{\beta} C_2'$. Then as $\beta \notin \mathsf{Q}^-$ by Q-closure, by Lemma 4.6 $C \Rightarrow \xrightarrow{\beta} C'$ and $C_2 \xrightarrow{\bar{u}} \simeq C'$. Now $M_1 \Rightarrow M_1'' = (\nu\tilde{v})(C \mid S)$, and by A-confluence $M_1'' \simeq M_1$. Further, $M_1'' \Rightarrow \xrightarrow{\rho} M_1' = (\nu\tilde{v})(C' \mid S)$ and $(M_1', M_2') \in \mathscr{B}$.

2. $M_2' \equiv (\nu\tilde{y})(C_2 \mid S_2'^\flat)$, where $S_2 \xrightarrow{\beta} S_2'$. Since $S_2 \in \mathscr{S}_\natural^\varnothing$, if $\beta = \tau$ then $S_2 \simeq S_2'$. With this observation the argument is similar to the above.

3. $M_2' \equiv (\nu\tilde{y})(C_2' \mid S_2'^\flat)$ and $\rho = \tau$ as $C_2 \xrightarrow{\gamma} C_2'$, $S_2 \xrightarrow{\bar{\gamma}} S_2'$, where γ comp $\bar{\gamma}$ and $\gamma \notin \mathsf{Q}^-$. Since $S_2 \in \mathscr{S}_\natural^\varnothing$, $\gamma \notin \mathsf{A}^+$. The argument is a combination of those above.

4. $M_2' \equiv (\nu\tilde{y})(C_2' \mid S_2'^\flat)$ and $\rho = \tau$ as $C_2 \xrightarrow{\bar{\gamma}} C_2'$, $S_2 \xrightarrow{\gamma} S_2'$, where $\bar{\gamma}$ comp γ and $\bar{\gamma} \in \mathsf{Q}^-$. By A-confluence $M_1 \simeq M_1'' = (\nu\tilde{v})(C \mid S)$, and $(M_1'', M_2) \in \mathscr{B}$. Since C is weak Q-confluent, by Lemma 4.6 there are two cases:

(a) $\bar{u} = \bar{v}\bar{\beta}\bar{w}$, $C_2 \xrightarrow{\bar{v}} \simeq C_3 \xrightarrow{\bar{\beta}} \simeq C_4 \xrightarrow{\bar{w}} \simeq C$, $C_2' \xrightarrow{\overline{v^*}} \simeq C_3'$, $C_3 \xrightarrow{\overline{\gamma^*}} \simeq C_3'$, $\bar{\beta} = \overline{\gamma^*}$, and $C_3' \simeq C_4$. Then, where $S_2 \xrightarrow{v} \simeq S_3 \xrightarrow{\beta} \simeq S_4 \xrightarrow{w} \simeq S$, since S is weak Q-confluent and Q-determinate, by Lemma 4.6 $S_2' \xrightarrow{v} \simeq S_4$. Hence $(M_1'', (\nu\tilde{x})(C_2' \mid S_2'^\flat)) \in \mathscr{B}$ as $C_2' \xrightarrow{\overline{vw}} \simeq C$ and $S_2' \xrightarrow{vw} \simeq S$.

(b) The previous case does not hold and $C \xrightarrow{\overline{\gamma^*}} \simeq C'$, $C_2' \xrightarrow{\overline{u^*}} \simeq C'$ where the bound name in $\bar{\gamma}$ does not occur in $\bar{u}$. Again by Lemma 4.6 either (i) $u = v\gamma w$, $S_2 \xrightarrow{v} \simeq S_3 \xrightarrow{\gamma} \simeq S_4 \xrightarrow{w} \simeq S$, and $S_2' \xrightarrow{v} \simeq S_4$, or (ii) $S \xrightarrow{\gamma} \simeq S'$ and $S_2 \xrightarrow{u} \simeq S'$. But (i) is impossible since the bound A-name of $\bar{\gamma}$ does not occur in $\bar{u}$, so that γ cannot occur in u. So assume (ii). Since $S_2' \in \mathscr{S}_\natural^a$, $S_2' \xrightarrow{\bar{\alpha}} \simeq S_2''$ or $S_2' \xrightarrow{\tau} \simeq \xrightarrow{\bar{\alpha}} \simeq S_2''$, where $\mathsf{subj}(\bar{\alpha}) = \bar{a}$, and since $C_2' \in \mathscr{C}_\natural^a$, $C_2' \xrightarrow{\alpha} \simeq C_2''$, where α comp $\bar{\alpha}$. By Theorem 4.19, $M_2 \simeq M_2'' = (\nu\tilde{w})(C_2'' \mid S_2''^\flat)$. Further, by Lemma 4.6 $S' \xrightarrow{\bar{\alpha}} \simeq S''$ of $S' \xrightarrow{\tau} \simeq \xrightarrow{\bar{\alpha}} \simeq S''$, where $S_2'' \xrightarrow{u} \simeq S''$, and $C' \xrightarrow{\alpha} \simeq C''$ where $C_2'' \xrightarrow{\overline{u^*}} \simeq C''$. Hence $M_1'' \Rightarrow M_1' = (\nu\tilde{v})(C'' \mid S'')$ and $(M_1', M_2'') \in \mathscr{B}$ and S'' is A^--inert. In summary, $M_1 \Rightarrow M_1'' \Rightarrow M_1'$, $M_1 \simeq M_1''$, and $(M_1'', M_2) \in \mathscr{B} \simeq$.

Now suppose $(M_1, M_2) \in \mathscr{B}$ and $M_1 \xrightarrow{\rho} M_1'$. There are several cases.

1. Suppose $M_1' \equiv (\nu\tilde{y})(C_1' \mid S_1)$, where $C_1 \xrightarrow{\beta} C_1'$. The case when $\beta = \tau$ and $C_1 \simeq C_1'$ is clear. Otherwise, by A-confluence, $C \Rightarrow \xrightarrow{\beta} C'$ and $C_1' \xrightarrow{s} \simeq C'$. Since C_2 is a QA-client, $C_2 \Rightarrow \xrightarrow{\beta^\dagger} C_2' \xrightarrow{\overline{u^\dagger}} \simeq C'$.

So $M_2 \Rightarrow \xrightarrow{\rho} M_2' = (\nu\tilde{w})(C_2' \mid S_2^\flat)$ and $(M_1', M_2') \in \mathscr{B}$.

2. Suppose $M_1' \equiv (\nu\tilde{y})(C_1 \mid S_1')$, where $S_1 \xrightarrow{\beta} S_1'$ and $\beta \neq \tau$. Then by A-confluence, $S \Rightarrow \xrightarrow{\beta} S'$ and $S_1' \xrightarrow{s} \simeq S'$. Since S_2 is a QA-server, $S_2 \Rightarrow \xrightarrow{\beta} S_2' \xrightarrow{u} \simeq S'$. So $M_2 \Rightarrow \xrightarrow{\rho} M_2' = (\nu\tilde{w})(C_2 \mid S_2'^\flat)$ and $(M_1', M_2') \in \mathscr{B}$.

3. Suppose $M_1' \equiv (\nu\tilde{y})(C_1 \mid S_1')$ and $S_1 \xrightarrow{\tau} S_1'$. The case $S_1' \simeq S_1$ is simple so suppose $S_1 \xrightarrow{\tau}_{\simeq} S_1'$. By A-confluence $S \Rightarrow \xrightarrow{\tau} S'$ and $S_1' \xrightarrow{\bar{s}}_{\simeq} S'$. Further since S is A^--inert, $S' \Rightarrow \xrightarrow{\bar{\alpha}} S''$, where $\bar{\alpha} \in \mathsf{A}^-$. Hence by commutativity, $S_1' \Rightarrow \xrightarrow{\overline{\alpha^\dagger}} S_1''$, and by A-confluence, $S_1'' \xrightarrow{\overline{s^\dagger}}_{\simeq} S''$. Note that S'' is A^--inert.

Since $S_2 \in \mathscr{S}^{\varnothing}_{\natural}$ and $S_2 \xrightarrow{u}_{\simeq} S$, where $u \in \mathsf{Q}^+*$, by commutativity for some $\beta \in u$, $S_2 \xrightarrow{\beta}_{\simeq} \xrightarrow{\tau}_{\simeq} S_2' \xrightarrow{t}_{\simeq} S'$, where t is u with β deleted. Hence by commutativity $S_2' \Rightarrow \xrightarrow{\bar{\alpha}} S_2''$. By Lemma 4.6, A-determinacy, and Q-determinacy, $S_2'' \xrightarrow{t}_{\simeq} S''$. Further, there is $\bar{\beta}\,\mathsf{comp}\,\beta$ such that $\bar{\beta} \in \bar{u}$ and as $C_2 \xrightarrow{\bar{u}}_{\simeq} C$, by commutativity, $C_2 \xrightarrow{\overline{\beta^\dagger}}_{\simeq} C_2' \xrightarrow{\overline{t^\dagger}}_{\simeq} C$, where $C_2' \in \mathscr{C}^a_{\natural}$. Hence $C_2' \xrightarrow{\alpha} C_2''$ where $\alpha\,\mathsf{comp}\,\bar{\alpha}$ and by Lemma 4.6, $C_2'' \xrightarrow{\overline{t^\dagger}}_{\simeq} C'$ and $C \Rightarrow \xrightarrow{\alpha} C'$. So, $M_2 \Rightarrow \to M_2' = (\nu\tilde{v})(C_2'' \mid S_2''^{\flat})$ and since $C_1 \xrightarrow{s\alpha}_{\simeq} C'$ and $S_1' \xrightarrow{\overline{s\bar{\alpha}}}_{\simeq} S''$, $(M_1', M_2') \in \mathscr{B}$.

4. If $\alpha = \tau$ and $M_1' = (\nu\tilde{y})(C_1' \mid S_1')$ where $C_1 \xrightarrow{\gamma} C_1'$, $S_1 \xrightarrow{\bar{\gamma}} S_1'$, and $\gamma \notin \mathsf{Q}^- \cup \mathsf{A}^+$ then the argument is a combination of those above.

5. Suppose $\alpha = \tau$ and $M_1' = (\nu\tilde{y})(C_1' \mid S_1')$, where $C_1 \xrightarrow{\bar{\gamma}} C_1'$ and $S_1 \xrightarrow{\gamma} S_1'$ with $\bar{\gamma}\,\mathsf{comp}\,\gamma$ and $\bar{\gamma} \in \mathsf{Q}^-$. By A-confluence, $C \xrightarrow{\bar{\gamma}}_{\simeq} C'$ and $C_1' \xrightarrow{s}_{\simeq} C'$, and $S \xrightarrow{\gamma}_{\simeq} S'$ and $S_1' \xrightarrow{\bar{s}}_{\simeq} S'$. If S' is A^--inert then $C_2 \xrightarrow{\overline{u\gamma}}_{\simeq} C'$ and $S_2 \xrightarrow{u\gamma}_{\simeq} S'$, so $(M_1', M_2) \in \mathscr{B}$. Otherwise, $S' \xrightarrow{\bar{\alpha}} S''$ for some $\bar{\alpha} \in \mathsf{A}^-$ and S'' is A^--inert. Since $S_2 \in \mathscr{S}^{\varnothing}_{\natural}$, S is A^--inert, and $S_2 \xrightarrow{u}_{\simeq} S \xrightarrow{\gamma}_{\simeq} S' \xrightarrow{\bar{\alpha}} S''$, by Lemma 4.18(3), $S_2 \xrightarrow{\gamma}_{\simeq} S_2' \xrightarrow{\bar{\alpha}}_{\simeq} S_2'' \xrightarrow{u}_{\simeq} S''$. Moreover, since $C_2 \xrightarrow{\bar{u}}_{\simeq} C \xrightarrow{\bar{\gamma}}_{\simeq} C'$, by commutativity, $C_2 \xrightarrow{\overline{\gamma^\dagger}}_{\simeq} C_2' \xrightarrow{\overline{u^\dagger}}_{\simeq} C'$. Further, $C_2' \xrightarrow{\alpha} C_2''$ where $\alpha\,\mathsf{comp}\,\bar{\alpha}$, and by Lemma 4.6, $C' \Rightarrow \xrightarrow{\alpha} C''$, where $C_2'' \xrightarrow{\overline{u^\dagger}}_{\simeq} C''$. Hence, $M_2 \Rightarrow M_2' = (\nu\tilde{w})(C_2' \mid S_2'^{\flat}) \Rightarrow M_2'' = (\nu\tilde{v})(C_2'' \mid S_2''^{\flat})$, where $S_2'' \xrightarrow{u^\dagger}_{\simeq} S''$ and $C_2'' \xrightarrow{\overline{u^\dagger}}_{\simeq} C''$. Since also $S_1' \xrightarrow{\overline{s\bar{\alpha}}}_{\simeq} S''$ and $C_1 \xrightarrow{s\alpha}_{\simeq} C''$, $(M_1', M_2'') \in \mathscr{B}$, and by A-confluence $M_2' \simeq M_2''$.

6. Finally, suppose $\alpha = \tau$ and $M_1' = (\nu\tilde{y})(C_1' \mid S_1'^{\flat})$, where $C_1 \xrightarrow{\gamma} C_1'$ and $S_1 \xrightarrow{\bar{\gamma}} S_1'$ with $\gamma \in \mathsf{A}^+$. Then $\mathsf{subj}(\gamma) \in \mathsf{subj}(s)$ as S is A^--inert. So by Lemma 4.6 and A-determinacy, $S_1' \xrightarrow{\overline{s^*}}_{\simeq} S$, where $\overline{s^*}$ is $\bar{s}$ with $\bar{\gamma}$ deleted and possibly change of bound names, and by Lemma 4.6 $C_1' \xrightarrow{s^*}_{\simeq} C$. Thus $(M_1', M_2) \in \mathscr{B}$.

We have shown that if $(M_1, M_2) \in \mathscr{B}$ and $M_1 \xrightarrow{\alpha} M_1'$ then $M_2 \Rightarrow M_2'' \xrightarrow{\alpha} M_2'$ where $(M_1, M_2''), (M_1', M_2') \in \mathscr{B} \simeq$, (or $\alpha = \tau$ and $(M_1', M_1) \in \mathscr{B} \simeq$), and vice versa. From this the result follows. ∎

4.2. *Analysis of* Q_0

In this section we apply the partial-confluence theory to prove first that $P_0 \simeq Q_0$ and then that $\mathscr{C}[Q_0] \simeq \mathscr{C}[Q_0^{\flat}]$ for each context $\mathscr{C}[\cdot]$ in a certain class. To do this we must examine P_0. We begin with T_0.

From the definitions we see that $\mathsf{NODE}\,\langle p, \tilde{k}, \tilde{p}\rangle$ may alternately accept a question via p and answer it using a name occurring in the question. We refer to $\mathsf{NODE}\,\langle p, \tilde{k}, \tilde{p}\rangle$ as a *quiescent node* and to an immediate derivative N of it as an *active node*, and we set $name(\mathsf{NODE}\,\langle p, \tilde{k}, \tilde{p}\rangle) = name(N) = p$. Similarly we define *quiescent leaf*, *active leaf*, and $name(L)$, where L is a leaf. We refer to $\mathsf{ROOT}\,\langle p, \tilde{k}, \tilde{p}\rangle$ as a *quiescent root*, to an immediate derivative R of it as an *active root*, and to a derivative R' of it of the form $(\nu p_0)\,\overline{put}(p_0).W$ as a *quasi-quiescent root*; we set $name(\mathsf{ROOT}\,\langle p, \tilde{k}, \tilde{p}\rangle) = name(R) = name(R') = p$.

The following lemma gives a rough description of the form of derivatives of T_0.

LEMMA 4.21. *A derivative of T_0 is of the form $(v\tilde{z})(S \mid \Pi_i C_i)$ with free names only of types $\updownarrow$ P, $\updownarrow\langle\updownarrow$ P, $\updownarrow\updownarrow$ P$\rangle$, D, $\updownarrow$ R, and $\updownarrow\updownarrow$ P, where S is a store, each C_i is a root, a node, or a leaf, and if $i \neq j$ then name$(C_i) \neq$ name(C_j).*

Proof. The proof is by induction on the length of computation and involves a straightforward case analysis, which is omitted. ▮

A derivative of T_0 is *active* if at least one of its components is, and *quiescent* otherwise. Below we will give a more detailed description of those derivatives, but first:

LEMMA 4.22. *T_0 is a prompt* A*-server where* $\mathsf{A} = \{\updownarrow \mathsf{R}\}$.

Proof. It is possible to give a typing system for agents that guarantees A-confluence; this may be done by extending the system introduced in [17]. Here, however, we argue directly.

That T_0 is A-confluent follows from the fact that each active component of a derivative of T_0 has a different free A-name via which it may immediately return a unique value. This observation shows also that T_0 is prompt. (Note that a quasi-quiescent root can only interact with the store.) That T_0 is an A-server is seen by examining its definition. If a component receives an A-name it uses it exactly once to return a value.

In more detail, we observe first that the agents NODE, ROOT, and LEAF are prompt A-servers. This is because whenever any of these agents receives an A-name it uses it immediately and exactly once to return a value. It is not difficult to prove that a composition of prompt A-servers, none of which uses A-names for input, is itself a prompt A-server. Since T_0 is the composition of a LEAF, a ROOT, and the store agent (which does not use A-names), and since also none of these agents ever uses an A-name for input, we may conclude that T_0 is a prompt A-server. ▮

We now examine the operation agents S_0 and I_0. A *searcher* is a derivative of $get(p).S\langle k, p, a_s\rangle$. We say a searcher is *active* if of the form $r(y).W$, and *quiescent* otherwise. Similarly an *inserter* is a derivative of $get(p).Down\langle k, b, a_i, p, \langle\,\rangle\rangle$. An inserter is *active* if of the form $r(y).W$, and *quiescent* otherwise. We refer to an inserter as being in its *down*, *insert*, or *up phase* with the obvious meanings.

LEMMA 4.23. *A derivative of $S_0 \mid I_0$ is of the form $S_0 \mid I_0 \mid \Pi_i A_i$ where each A_i is a searcher or an inserter.*

Proof. Immediate from the definitions. ▮

A derivative of $S_0 \mid I_0$ is *active* if at least one of its components is, and *quiescent* otherwise. To complement Lemma 4.22 we have:

LEMMA 4.24. *$S_0 \mid I_0$ is an* A*-client, where* $\mathsf{A} = \{\updownarrow \mathsf{R}\}$.

Proof. That $S_0 \mid I_0$ is A-confluent follows from the observation that when a searcher or inserter interrogates a node it supplies a fresh A-name, waits to receive

a value via that name, and then proceeds as determined by that value. That it is an A-client follows similarly. ▮

Recalling the definition of $T_0^\flat$ where $\mathsf{Q} = \{\updownarrow \mathsf{P}\}$ and $\mathsf{A} = \{\updownarrow \mathsf{R}\}$ we set

$$Q_0 \stackrel{\text{df}}{=} (v\mathit{get}\ \mathit{next})(S_0 \mid I_0 \mid T_0^\flat).$$

We now prove equivalence (1):

THEOREM 4.25. $Q_0 \simeq P_0$.

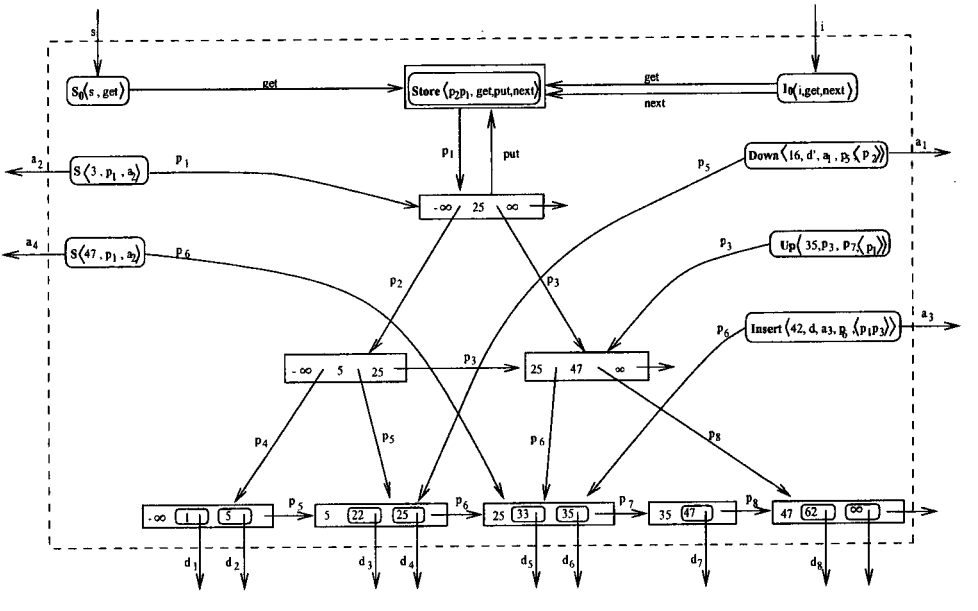
Proof. Since $S_0 \mid I_0$ is an A-client, T_0 is a prompt A-server, and neither has any free A-names, the result follows by Theorem 4.15 and the observation that P_0 is A-closed, which is a consequence of the fact that P_0 is well-typed: by the typing, no derivative of P_0 has a free A-name. ▮

A derivative Q of Q_0 is of the form $(v\tilde{z})(Z \mid T)$ where Z and T are both quiescent or both active; in the former case Q is *quiescent*, in the latter *active*. If Q is active its two active agents share an A-name. By Lemma 4.8 the $\simeq$-state of an A-closed composition of A-confluent agents is not changed by an interaction via an A-name. Hence,

LEMMA 4.26. *If Q is an active derivative of Q_0 then there is a unique quiescent Q' such that $Q \xrightarrow{\tau} Q'$; and $Q' \simeq Q$.*

Proof. The two active components of Q may interact in exactly one way via the A-name which they share and thereby evolve to a quiescent Q'. By Lemma 4.8, $Q' \simeq Q$. ▮

Hence we can focus on quiescent derivatives of Q_0. The following figure shows the structure of one.



Let $Q=(v\tilde{z})(Z \mid T)$ be a quiescent derivative of Q_0. Then $T=(vz)(S \mid \Pi_{i\in I}C_i)$ where $S=\mathsf{STORE}\langle\langle q_1^* \cdots q_\ell^*\rangle \cdots\rangle$ or $\vec{n}(q_h^*).\mathsf{STORE}\langle\langle q_1^* \cdots q_\ell^*\rangle \cdots\rangle$, $C_i=\mathsf{NODE}\langle p_i, \tilde{k}_i, \tilde{p}_i\rangle$ if C_i is a node, $C_i=\mathsf{LEAF}\langle p_i, \tilde{k}_i, \tilde{b}_i, q_i\rangle$ if C_i is a leaf, and $C_i=\mathsf{ROOT}\langle p_i, \tilde{k}_i, \tilde{p}_i, put\rangle$ or

$$C_i\langle p_i, \tilde{k}_i, \tilde{p}_i, put\rangle=(vp_0)\,\overline{put}(p_0).(vp')(\mathsf{ROOT}\langle p, \tilde{k}, \tilde{p}\rangle \mid \mathsf{NODE}\langle p_1, \tilde{k}_{i1}, \tilde{p}_{i1}p'\rangle \mid \mathsf{NODE}\langle p', \tilde{k}_{i2}, \tilde{p}_{i2}\rangle),$$

where $\tilde{k}_i=\tilde{k}_{i1}\tilde{k}_{i2}$ with the duplicate k_{m+1} deleted and $\tilde{p}_i=\tilde{p}_{i1}\tilde{p}_{i2}$. Let $N=\{p_i \mid i\in I\}$ and define $\rightsquigarrow$ on N by:

1. if C_i is a node or the root then $p_i \rightsquigarrow p$ if $p\in\tilde{p}_i$;
2. if C_i is a leaf then $p_i \rightsquigarrow p$ if $p=q_i$.

Thus $\rightsquigarrow$ is intended to give the "points to" relation among the cells of the data structure. The following long lemma establishes some invariant properties satisfied by the quiescent derivatives of Q_0.

LEMMA 4.27. *In the notation above*:

1. *Exactly one C_i is a root, and if C_i is the root then $\tilde{k}_i=k_{i1}\cdots k_{im_i}$ and $\tilde{p}_i=p_{i1}\cdots p_{im_i}$ where $2\leqslant m_i\leqslant 2m+1$ if the root is quiescent and $m_i=2m+2$ if it is quasi-quiescent; $k_{i1}=-\infty$, $k_{im_i}=\infty$, and $p_{im_i}=nil$; if $h<m_i$ then $p_{ih}\in N$, say $p_{ih}=p_j$, and $k_{j1}=k_{ih}$ and $k_{jm_j}\leqslant k_{i(h+1)}$.*
2. *If C_i is a node then: $\tilde{k}_i=k_{i1}\cdots k_{im_i}$ and $\tilde{p}_i=p_{i1}\cdots p_{im_i}$ where $m+1\leqslant m_i\leqslant 2m+1$; $k_{im_i}=\infty$ iff $p_{im_i}=nil$; if $k_{ih}\neq\infty$ then $p_{ih}\in N$, say $p_{ih}=p_j$, and $k_{j1}=k_{ih}$ and if $h<m_i$ then $k_{jm_j}\leqslant k_{i(h+1)}$, and if $h=m_i$ then C_j is a node.*
3. *If C_i is a leaf then: $\tilde{k}_i=k_{i1}\cdots k_{im_i}$ and $\tilde{b}_i=b_{i2}\cdots b_{im_i}$ where $2\leqslant m_i\leqslant 2m+1$; $k_{im_i}=\infty$ iff $q_i=nil$; if $k_{im_i}\neq\infty$ then $q_i\in N$, say $q_i=p_j$, and $k_{j1}=k_{im_i}$ and C_j is a leaf.*
4. *If $h<m_i$ then $k_{ih}<k_{i(h+1)}$.*
5. *If $p_i\rightsquigarrow p_j$ then $k_{i1}\leqslant k_{j1}$.*
6. *$(N, \rightsquigarrow)$ is acyclic.*
7. *In the store, $\ell\geqslant 1$, and if $1\leqslant h\leqslant\ell$ then $q_h^*\in N$, say $q_h^*=p_{i(h)}$, and $k_{i(h)\,1}=-\infty$. Also: $C_{i(\ell)}$ is the root; if $h<\ell$ then $C_{i(h)}$ is a node; if $1<h\leqslant\ell$ then $p_{i(h)\,1}=p_{i(h-1)}$; $p_{i(1)\,1}=p_j$ where C_j is a leaf and $k_{j1}=-\infty$.*
8. *If Z has a component $S\langle k, p, a_s\rangle$ then for some i, $p=p_i$ and $k>k_{i1}$.*
9. *If Z has a component $Down\langle k, b, a_i, q_0, \langle q_1\cdots q_n\rangle\rangle$ then: $\{q_0, ..., q_n\}\subseteq N$, say $q_h=p_{i(h)}$; if $j\geqslant 1$ then $C_{i(j)}$ is not a leaf; $k_{i(n)\,1}\leqslant\cdots\leqslant k_{i(1)\,1}\leqslant k_{i(0)\,1}<k$; if $n=0$ then $q_0=q_h^*$ for some h; if $n\geqslant 1$ then $q_n=q_h^*$ for some h.*

10. *If Z has a component Insert* $\langle k, b, a_i, q_0, \langle q_1 \cdots q_n \rangle \rangle$ *then*: $\{q_0, ..., q_n\} \subseteq N$, *say* $q_h = p_{i(h)}$; *if* $j \geqslant 1$ *then* $C_{i(j)}$ *is not a leaf*; $C_{i(0)}$ *is a leaf*; $k_{i(n)\,1} \leqslant \cdots \leqslant k_{i(1)\,1} \leqslant k_{i(0)\,1} < k$; $n \geqslant 1$ *and* $q_n = q_h^*$ *for some* h.

11. *If Z has a component Up* $\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle \rangle$ *or* $\overline{a_i}.Up\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle \rangle$ *then*: $\{q_0, ..., q_n\} \subseteq N$, *say* $q_h = p_{i(h)}$; *if* $j \geqslant 1$ *then* $C_{i(j)}$ *is not a leaf*; $k_{i(n)\,1} \leqslant \cdots \leqslant k_{i(1)\,1} < k = k_{i(0)\,1}$; $n \geqslant 2$ *and* $q_n = q_h^*$ *for some* h.

12. *If Z has a component Up'* $\langle k, q_0, q_1, \langle q_2 \rangle \rangle$ *then*: $\{q_0, q_1, q_2\} \subseteq N$, *say* $q_h = p_{i(h)}$; $C_{i(1)}$ *and* $C_{i(2)}$ *are not leaves*; $k_{i(2)\,1} \leqslant k_{i(1)\,1} < k = k_{i(0)\,1}$; $q_2 = q_h^*$ *for some* h.

13. *If Z has a component* $(\nu n)\,\overline{next}(\langle q_1, n \rangle).n(q_2).Up'\langle k, q_0, q_2, \langle q_2 \rangle \rangle$ *then*: $\{q_0, q_1\} \subseteq N$, *say* $q_h = p_{i(h)}$; $k_{i(1)\,1} < k = k_{i(0)\,1}$; $q_1 = q_h^*$ *for some* $h < \ell$.

Proof. The proof is by induction on the length of the computation. It is straightforward to check that the conditions hold of Q_0. Assume they hold of Q which is quiescent and consider $Q \xrightarrow{\alpha} Q'$. There are 11 cases. Although the proof is long, the use of the model introduced in this paper helps greatly in managing the detail (at least some of which must surely be handled in any proof).

- A. Q has a quasi-quiescent root and Q' a quiescent root.
- B. An operation is requested: Q' has a $get(p).S\langle k, p, a_s \rangle$ or $get(p).Down\langle k, b, a_i, p, \langle\,\rangle\rangle$ which Q does not.
- C. A search begins: Q has $get(q).S\langle k, q, a_s \rangle$ and Q' has $S\langle k, p, a_s \rangle$.
- D. An insertion begins: Q has $get(q).Down\langle k, b, a_i, q, \langle\,\rangle\rangle$ and Q' has $Down\langle k, b, a_i, p, \langle\,\rangle\rangle$.
- E. A searcher $S\langle k, p, a_s \rangle$ requests a search at p.
- F. A search finishes: Q has $\overline{a_s}(b).\mathbf{0}$ and Q' has $\mathbf{0}$.
- G. An inserter $Down\langle k, b, a_i, q_0\langle q_1 \cdots q_n \rangle\rangle$ requests a search at q_0.
- H. An inserter $Insert\langle k, b, a_i, q_0, \langle q_1 \cdots q_n \rangle\rangle$ requests an insertion at q_0.
- I. An insertion finishes: Q has $\overline{a_i}.\mathbf{0}$ and Q' has $\mathbf{0}$, or Q has $\overline{a_i}.Up\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle\rangle$ and Q' has $Up\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle\rangle$.
- J. An inserter $Up\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle\rangle$ requests an add at q_1.
- K. An inserter $Up'\langle k, q_0, q_1, \langle q_2 \cdots q_n \rangle\rangle$ requests an add at q_1.

First note that if the root is quasi-quiescent then no searcher or inserter can interact with it. Further, if a searcher or inserter interacts with another cell, the return of the result commutes with the action of the root becoming quasi-quiescent, even in case K when Up' reads the store. The proof is mostly a matter of routine checking. The most significant cases are: A1, A2, A4, A5, A6, A7, C8, D9, E8, G9, G10, H10, H11, I11, J5, J6, J11, J12, K5, K6, K12, K13. Of these J5, J6, K5, K6 are most notable. The add results in the new pointing $p_{i(1)} \rightsquigarrow p_{i(0)}$, and before the inserter requests the add at q_1 we know $k_{i(n)\,1} \leqslant \cdots \leqslant k_{i(1)\,1} < k = k_{i(0)\,1}$. If the add were to introduce a cycle it would be that $p_{i(1)} \rightsquigarrow p_{i(0)} \rightsquigarrow \cdots \rightsquigarrow p_{i(1)}$, so $k_{i(1)\,1} \leqslant k_{i(0)\,1} \leqslant \cdots \leqslant k_{i(1)\,1}$, so $k_{i(1)\,1} = k_{i(0)\,1}$, which is not the case. ∎

As a consequence we have the vital lemma that a searcher or inserter can always reach its target leaf;

LEMMA 4.28. *In the notation above, suppose Z has a component* $get(q).S\langle k, q, a_s\rangle$, $S\langle k, p, a_s\rangle$, $get(q).Down\langle k, b, a_i, q, \langle\ \rangle\rangle$, $Down\langle k, b, a_i, p, \tilde{q}\rangle$, *or* $Insert\langle k, b, a_i, p, \tilde{q}\rangle$.

1. *There is a unique leaf* $\mathsf{LEAF}\langle p', \tilde{k}, \tilde{b}, q'\rangle$ *such that* $k_1 < k \leqslant k_j$ *where* $\tilde{k} = k_1 \cdots k_j$.

2. *Further,* $Q \Rightarrow Q' = (v\tilde{z})(Z' \mid T)$ *where* Z' *differs from* Z *only in that the component in question is replaced by* $S\langle k, p', a_s\rangle$ *or* $Insert\langle k, b, a_i, p', \tilde{q}'\rangle$.

Proof. From Lemma 4.27 it follows that there are $n \geqslant 1$ leaves $C_{i(1)}, \ldots, C_{i(n)}$ with $k_{i(1)\,1} = -\infty$, $k_{i(h)\,j} < k_{i(h)(j+1)}$ and $k_{i(h)\,m_{i(h)}} = k_{i(h+1)\,1}$ for $h < n$ and $j < i(h)_{m_{i(h)}}$, and $k_{i(n)\,m_{i(n)}} = \infty$. The first part follows.

Again from Lemma 4.27 ((6) and (8)) we have that $(N, \rightsquigarrow)$ is acyclic and, if in $S\langle k, p, a_s\rangle$, $p = p_i$, then $k > k_{i1}$ and hence the searcher may proceed, and similarly in the other cases. ▮

Now let $\mathsf{Q} = \{\updownarrow\langle \mathsf{int}, \updownarrow \mathsf{D}\rangle, \updownarrow\langle \mathsf{int}, \updownarrow \mathsf{D}, \updownarrow\langle\ \rangle\rangle\}$ and $\mathsf{A} = \{\updownarrow \mathsf{D}, \updownarrow\langle\ \rangle\}$. We have:

THEOREM 4.29. Q_0 *is an almost-prompt* QA*-server.*

Proof. That $Q_0^{\natural}$ is A-confluent follows from the fact that each searcher or inserter uses its free A-name just once to return the value it receives from the data structure. That it is Q-confluent follows immediately from the definition. To prove the other conditions we have a lemma, preceded by a definition:

DEFINITION 4.30. Suppose $T = (v\tilde{z})(S \mid \Pi_i C_i)$ is a quiescent derivative of T_0. Define val_T: $\mathsf{int} \to \mathsf{D}$ as follows. Given k let $C_i = \mathsf{LEAF}\langle p, \tilde{k}, \tilde{b}, q\rangle$ be the unique leaf of T such that $k_1 < k \leqslant k_j$ where $\tilde{k} = k_1 \cdots k_j$. Then set $val_T(k) = b_h$ if $k = k_h$ and *nil* if $k \notin \tilde{k}$.

LEMMA 4.31. *Define* $\mathscr{B}$ *by setting* $(Q, Q') \in \mathscr{B}$ *if* $Q = (v\tilde{z})(Z \mid T)$ *and* $Q' = (v\tilde{z})(Z' \mid T')$ *are quiescent derivatives of* Q_0 *in* $\mathscr{Q}_{\natural}$ *(where* $\mathscr{Q}$ *is the transition system of* Q*) such that*

1. $val_T = val_{T'}$,

2. Z *has a component* $\overline{a_s}(b).\mathbf{0}$ *iff* Z' *has the same component,*

3. Z *has a component* $\overline{a_i}.W$ *iff* Z' *has a component* $\overline{a_i}.W'$,

4. Z *has a component* $get(q).S\langle k, q, a_s\rangle$ *or* $S\langle k, p, a_s\rangle$ *iff* Z' *has a component* $get(q).S\langle k, q, a_s\rangle$ *or* $S\langle k, p', a_s\rangle$, *and*

5. Z *has a component* $get(q).Down\langle k, b, a_i, q, \langle\ \rangle\rangle$ *or* $Down\langle k, b, a_i, p, \tilde{q}\rangle$ *or* $Insert\langle k, b, a_i, p, \tilde{q}\rangle$ *iff* Z' *has a component* $get(q).Down\langle k, b, a_i, q, \langle\ \rangle\rangle$ *or* $Down\langle k, b, a_i, p', \tilde{q}'\rangle$ *or* $Insert\langle k, b, a_i, p', \tilde{q}'\rangle$.

Then $(Q, Q') \in \mathscr{B}$ *implies* $Q \simeq_{\mathsf{A}} Q'$.

Proof. Note that if $(Q, Q') \in \mathscr{B}$ then in Z and Z' the same results are available for immediate return and the same search and insertion operations must be in

progress. The operations may, however, be at different stages. In addition, Q and Q' may have arbitrary inserters in up phase. Suppose $(Q, Q') \in \mathscr{B}$ and $Q \xrightarrow{\alpha} Q_1$.

If α is $s\langle k, a_s\rangle$, $i\langle k, b, a_i\rangle$, $\overline{a_s}b$ or $\overline{a_i}$, it is immediate from the definition that $Q' \xrightarrow{\alpha} Q'_1$ with $(Q_1, Q'_1) \in \mathscr{B}$.

If α is τ and is not an interaction between either $S\langle k, p, a_s\rangle$ and $\mathsf{LEAF}\langle p, \tilde{k}, \tilde{b}, q\rangle$ or $Insert\langle k, b, a_i, p, \tilde{q}\rangle$ and $\mathsf{LEAF}\langle p, \tilde{k}, \tilde{b}, q\rangle$ where $k_1 < k \leqslant k_j$ where $\tilde{k} = k_1 \cdots k_j$, then Q' may match by doing nothing: where $Q'_1 \xrightarrow{\tau} Q''_1$ with Q''_1 quiescent, $Q'_1 \simeq Q''_1$ and $(Q''_1, Q') \in \mathscr{B}$.

If Q_1 results from Q by interaction between $S\langle k, p, a_s\rangle$ and $\mathsf{LEAF}\langle p, \tilde{k}, \tilde{b}, q\rangle$, where $k_1 < k \leqslant k_j$ with $\tilde{k} = k_1 \cdots k_j$, then T is unchanged and Q_1 has $\overline{a_s}(b).\mathbf{0}$, where $b = val_T(k)$ and Q has $S\langle k, p, a_s\rangle$. By the previous lemma, the corresponding component of Q' can progress to the corresponding leaf of T' and then make the corresponding step, leaving T' unchanged and yielding Q'_1 with $\overline{a_s}(b).\mathbf{0}$ (as $val_T = val_{T'}$) in place of that component. Then $(Q_1, Q'_1) \in \mathscr{B}$.

The argument in the case of insertion is similar with the observation that where T becomes T_1 and T' becomes T'_1, $val_{T_1} = val_{T'_1}$. ▮

It is clear that the interactions in the last two cases above are decisive; i.e., $Q \not\simeq Q_1$. (In the case of a search for instance, in Q an insertion could be requested with key k and a value $b' \neq b$, and could overtake the search, and the $\bar{a}b'$ would be possible for $a \Rightarrow$-derivative of Q but not for any of Q_1.) It follows from the lemma that these are the only decisive τ-actions. The four conditions of Definition 4.17 follow from these observations and the analysis above, recalling that $P_0 \simeq Q_0$. This completes the proof of Theorem 4.29. ▮

It is possible to extend this analysis to show that $Q_0 \simeq_{\mathsf{A}} B_0$. However, for the reasons explained earlier we show this result using the partial-confluence theory. From Theorem 4.20 we have equivalence (5) stated in the introduction to Section 4:

COROLLARY 4.32. *Let $\mathscr{C}[\cdot] = (\nu\tilde{z})(C \mid \cdot)$, where C is a* QA*-client and $\mathscr{C}[Q_0]$ is* QA*-closed. Then $\mathscr{C}[Q_0] \simeq \mathscr{C}[Q_0^\flat]$.*

The agent B_0 is easy to comprehend:

THEOREM 4.33. *B_0 is an almost-prompt* QA*-server.*

Proof. From the definition it is immediate that $B_0^\natural$ is A-confluent and Q-confluent. That it enjoys properties 1–4 of Definition 4.17 is easily checked. Consider, for instance, Property 2. If $B = B\langle f, \sigma, \sigma^c, \imath, \imath^c\rangle$ then

1. $\sigma = \{\langle k, a_s\rangle\}$ and $\sigma^c = \imath = \imath^c = \varnothing$, and $B \xrightarrow{\tau} B' \xrightarrow{\overline{a_s}b}$, where $b = f(k)$, or
2. $\sigma^c = \{\langle a_s, b\rangle\}$ and $\sigma = \imath = \imath^c = \varnothing$, and $B \xrightarrow{\overline{a_s}b}$, or
3. $\imath = \{\langle k, b, a_i\rangle\}$ and $\sigma = \sigma^c = \imath^c = \varnothing$, and $B \xrightarrow{\tau} B' \xrightarrow{\overline{a_i}}$, or
4. $\imath^c = \{a_i\}$ and $\sigma = \sigma^c = \imath = \varnothing$, and $B \xrightarrow{\overline{a_i}}$. ▮

Hence, again by Theorem 4.20, we have equivalence (4) stated in the introduction to Section 4:

COROLLARY 4.34. *Let $\mathscr{C}[\cdot] = (\nu\tilde{z})(C \mid \cdot)$ where C is a* QA*-client and $\mathscr{C}[B_0]$ is* QA*-closed. Then $\mathscr{C}[B_0] \simeq \mathscr{C}[B_0^\flat]$.*

To complete the proof of Theorem 4.1 it remains to prove equivalences (3) and (2) stated in the introduction to Section 4. The first is straightforward:

THEOREM 4.35. $Q_0^\flat \simeq B_0^\flat$.

Proof. Define $\mathscr{S}$ by $(Q, B) \in \mathscr{S}$ if $Q = (v\tilde{z})(S_0 \mid I_0 \mid Z \mid T)$ and $B = B\langle f, \sigma, \sigma^c, \iota, \iota^c\rangle$ where $val_T = f$ and

1. Z is empty or is an inserter in its up phase, and $\sigma = \sigma^c = \iota = \iota^c = \varnothing$,
2. Z is a quiescent or active derivative of $S\langle k, p, a_s\rangle$, and $\sigma = \{\langle k, a_s\rangle\}$ and $\sigma^c = \iota = \iota^c = \varnothing$,
3. Z is a done searcher $\overline{a_s}(b).\mathbf{0}$, and $\sigma^c = \{\langle a_s, b\rangle\}$ and $\sigma = \iota = \iota^c = \varnothing$,
4. Z is a quiescent or active derivative of $Down\langle k, b, a_i, p, \langle\ \rangle\rangle$ in its down or insert phase, and $\iota = \{\langle k, b, a_i\rangle\}$ and $\sigma = \sigma^c = \iota^c = \varnothing$, or
5. Z is of the form $\overline{a_i}.\mathbf{0}$ or $\overline{a_i}.U$, and $\iota^c = \{a_i\}$ and $\sigma = \sigma^c = \iota = \varnothing$.

It is routine to check, using Lemma 4.28, that $\mathscr{S} \cup \simeq$ is a branching bisimulation. ∎

By this analysis we see that a variant system in which an inserter does not signal completion until it has completed its up phase is equivalent to the one studied.

We now have the final step in the proof of Theorem 4.1. We construct a QA-client C^* such that setting $\mathscr{C}^*[\cdot] = (vs, i)(C^* \mid \cdot)$, from $\mathscr{C}^*[Q_0] \simeq \mathscr{C}^*[B_0]$ we can deduce $Q_0 \simeq_{\mathsf{A}} B_0$. To do this we introduce the following types:

$$\mathscr{E} \equiv [search\_\langle \mathsf{int}, \updownarrow \mathscr{O}\rangle, insert\_\langle \mathsf{int}, \mathsf{D}, \updownarrow \mathscr{O}\rangle]$$

$$\mathscr{O} \equiv [datum\_\mathsf{D}, done\_\langle\ \rangle].$$

In the following, e: $\updownarrow \mathscr{E}$, o: $\updownarrow \mathscr{O}$, and $w : \mathscr{E}$. Define

$$C^*(e) \stackrel{\text{df}}{=} !\, e(w).\mathsf{case}(w)[search\_\langle k, o\rangle : (va_s)\, \bar{s}\langle k, a_s\rangle.a_s(b).\bar{o}(datum\_b).\mathbf{0},$$
$$insert\_\langle k, b, o\rangle : (va_i)\, \bar{\iota}\langle k, b, a_i\rangle.a_i().\bar{o}(done\_\langle\ \rangle).\mathbf{0}].$$

It is clear that $C^*\langle e\rangle$ is a QA-client. Hence with $\mathscr{C}^*[\cdot]$ as above, $\mathscr{C}^*[Q_0] \simeq \mathscr{C}^*[Q_0^\flat] \simeq \mathscr{C}^*[B_0^\flat] \simeq \mathscr{C}^*[B_0]$.

THEOREM 4.36. $Q_0 \simeq_{\mathsf{A}} B_0$.

Proof. Set $(Q, B) \in \mathscr{S}$ if $(v\tilde{z})(C^* \mid R \mid Q) \simeq (v\tilde{z})(C^* \mid R \mid B)$, where $\mathrm{fn}(Q)^{\mathsf{A}} = \mathrm{fn}(B)^{\mathsf{A}} = F_s \cup F_i$, $R = R_s \mid R_i$, with the o_{a_s} and o_{a_i} distinct,

$$R_s = \Pi_{a_s \in F_s} a_s(b).\overline{o_{a_s}}(datum\_b).\mathbf{0} \qquad \text{and} \qquad R_i = \Pi_{a_i \in F_i} a_i(\ ).\overline{o_{a_i}}(done\_\langle\ \rangle).\mathbf{0}.$$

By the observations above it suffices to show that $\mathscr{S} \subseteq \simeq_{\mathsf{A}}$.

LEMMA 4.37. *Let* $M = (\nu\tilde{z})(C^* \mid R \mid Q)$ *and* $N = (\nu\tilde{z})(C^* \mid R \mid B)$.

1. *Suppose* $Q \xrightarrow{\tau} Q'$ *and* $M' = (\nu\tilde{z})(C^* \mid R \mid Q')$. *If* $Q' \not\simeq Q$ *then* $M' \not\simeq M$.
2. *Suppose* $B \xrightarrow{\tau} B'$ *and* $N' = (\nu\tilde{z})(C^* \mid R \mid B')$. *Then* $B' \not\simeq B$ *and* $N' \not\simeq N$.

Proof. We consider just 2 since the same ideas, together with the results above, show 1. From the definition of B_0 we see that B' differs from B in that the σ^c of B' contains $\langle a_s, b\rangle$ while the σ of B contains $\langle k, a_s\rangle$, or the ι^c of B' contains a_i while the ι of B contains $\langle k, b, a_i\rangle$. In the first case M can accept a request to insert k, b' where $b' \neq b$, carry out the insertion followed by the search, and then return b' as the result via o_{a_s}; and M' cannot match this. In the second M can accept a request to insert k, b', where $b' \neq b$, carry out this new insertion followed by the one in question, then signal completion of both leaving b the value of k rather than b'; and M' cannot match this. ∎

Suppose $(Q, B) \in \mathscr{S}$ and M, N are as in the lemma. We consider just the search operation; the insert operation is handled similarly.

1. Suppose $Q \xrightarrow{s\langle k, a_s\rangle}_{\natural} Q'$. Then $M \xrightarrow{\alpha} M' \xrightarrow{\tau} M''$, where $\alpha = e\,search\_\langle k, o\rangle$ with o fresh and for a fresh a_s, $M'' = (\nu\tilde{z}a_s)(C^* \mid R' \mid Q')$. Now, $N \xrightarrow{\alpha} N' \xrightarrow{\tau} N''$, where $N'' = (\nu\tilde{z}a_s)(C^* \mid R' \mid B')$ with $B \xrightarrow{s\langle k, a_s\rangle}_{\natural} B'$. It follows from the lemma and A-confluence that $N' \simeq M'$. But by Q-confluence, $M'' \simeq M'$ and $N'' \simeq N'$. Hence $(Q', B') \in \mathscr{S}$. The converse is similar.

2. Suppose $Q \xrightarrow{\overline{a_s}b}_{\natural} Q'$. Then $M \xrightarrow{\tau} M' \xrightarrow{\alpha} M''$, where $\alpha = \overline{o_{a_s}}\,datum\_b$ and $M'' = (\nu\tilde{z}')(C^* \mid R' \mid Q')$. Since $M' \simeq M$ by A-confluence, from the lemma it follows that $N \xrightarrow{\tau} N' \xrightarrow{\alpha} N''$, where $N'' = (\nu\tilde{z}')(C^* \mid R' \mid B')$ and $B \xrightarrow{\alpha} B'$. Again by A-confluence, $N \simeq N'$, and again by the lemma $M'' \simeq N''$ and so $(Q', B') \in \mathscr{S}$. The converse is similar.

3. Suppose $Q \xrightarrow{\tau} Q'$. If $Q' \simeq Q$ then clearly $(Q', B) \in \mathscr{S}$. Suppose $Q' \not\simeq Q$ so that by the lemma, $M' \not\simeq M$, where $M' = (\nu\tilde{z}')(C^* \mid R \mid Q')$. Then $N \xrightarrow{\tau} N' = (\nu\tilde{z}')(C^* \mid R \mid B') \simeq M'$, where $B \xrightarrow{\tau} B'$. Then $(Q', B') \in \mathscr{S}$. The converse is similar.

This completes the proof of the theorem and hence of Theorem 4.2. ∎

4.3. Alternative Models

In the analysis of P_0, appeal was often made to the fact that the low key of a cell coincides with the high key of its left neighbour (if it has one). This duplication is not necessary, however, for the correctness of the algorithms: it is straightforward to show $T_0' \simeq T_0$ (which implies $P_0' \simeq P_0$) where T_0' and P_0' are variants of T_0 and P_0, respectively, without the duplication (we omit the definitions). This involves showing that the two tree variants, exhibit exactly the same behaviour by building the appropriate branching bisimulation.

As mentioned earlier, the process-calculus description of the algorithms is at a higher level of abstraction than those in [6, 16]. A process-calculus model with explicit representation of locking and unlocking can easily be given. Rather than write out the entire description we give just one definition and invite the interested

reader to work out the rest. The main differences are that cells become more passive in nature, and operation-agents become responsible for comparisons between keys and for creating cells when splitting is required (which they do by interacting with a replicator which may generate cells). A leaf may be either unlocked (LEAF) or locked (LEAF^{ℓ}):

$$\begin{aligned}&\mathsf{LEAF}(p, \tilde{k}, \tilde{b}, q)\\ &\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[\mathit{search}\_r : \bar{r}(\mathit{leaf}\_\langle \tilde{k}, \tilde{b}, q\rangle).\mathsf{LEAF}\,\langle p, \tilde{k}, \tilde{b}, q\rangle,\\ &\qquad\qquad \mathit{lock}\_r \quad : \bar{r}(\mathit{leaf}\_\langle \tilde{k}, \tilde{b}, q\rangle).\mathsf{LEAF}^{\ell}\,\langle p, \tilde{k}, \tilde{b}, q\rangle\,]\end{aligned}$$

$$\begin{aligned}&\mathsf{LEAF}^{\ell}(p, \tilde{k}, \tilde{b}, q)\\ &\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z)[\mathit{search}\_r : \bar{r}(\mathit{leaf}\_\langle \tilde{k}, \tilde{b}, q\rangle).\mathsf{LEAF}\,\langle p, \tilde{k}, \tilde{b}, q\rangle,\\ &\qquad\qquad \mathit{lock}\_r : \bar{r}(\mathit{locked}\_\langle\,\rangle).\mathsf{LEAF}^{\ell}\,\langle p, \tilde{k}, \tilde{b}, q\rangle,\\ &\qquad\qquad \mathit{unlock}\_\langle \tilde{k}', \tilde{b}', q'\rangle : \mathsf{LEAF}\,\langle p, \tilde{k}', \tilde{b}', q'\rangle\,]\end{aligned}$$

A reader who works out the model may care to examine its relationship to that studied in the paper and thus to establish the correctness of the algorithms as expressed in it. This route is easier than a direct analysis of the new model.

5. DELETION AND COMPRESSION

In this section we first consider the deletion and compression algorithms of [16], and then motivate and describe a new compression algorithm and briefly discuss its correctness.

The definitions of NODE, LEAF, and ROOT must be changed to accommodate the additional operations. In particular, the representation of a pointer to a node becomes a name of type $\updownarrow \mathsf{P}$, where

$$\begin{aligned}\mathsf{P} \equiv [&\mathit{search}\_\langle \mathsf{int}, \updownarrow \mathsf{R}\rangle, \mathit{insert}\_\langle \mathsf{int}, \mathsf{D}, \updownarrow \mathsf{R}\rangle, \mathit{add}\_\langle \mathsf{int}, \updownarrow \mathsf{P}, \updownarrow \mathsf{R}\rangle,\\ &\mathit{delete}\_\langle \mathsf{int}, \updownarrow \mathsf{R}\rangle, \mathit{compress}\_\updownarrow \mathsf{R}, \mathit{findp}\_\langle \mathsf{int}, \updownarrow \mathsf{P}, \updownarrow \mathsf{R}\rangle\,]\\ \mathsf{R} \equiv [&\mathit{link}\_\updownarrow \mathsf{P}, \mathit{nonlink}\_\updownarrow \mathsf{P}, \mathit{datum}\_\mathsf{D}, \mathit{done}\_\langle\,\rangle, \mathit{split}\_\langle \updownarrow \mathsf{P}, \mathsf{int}\rangle, \mathit{empty}\_\langle\,\rangle,\\ &\mathit{data}\_\langle \mathsf{int}^{2m+1}, \mathsf{D}^{2m-1}, \updownarrow \mathsf{P}, \updownarrow \mathsf{U}\rangle, \mathit{contents}\_\langle \mathsf{int}^{2m+1}, (\updownarrow \mathsf{P})^{2m+1}, \updownarrow \mathsf{U}\rangle,\\ &\mathit{right}\_\langle \updownarrow \mathsf{P}, \updownarrow \mathsf{U}\rangle, \mathit{left}\_\langle \updownarrow \mathsf{P}, \updownarrow \mathsf{U}\rangle, \mathit{retry}\_\langle\,\rangle\,]\\ \mathsf{U} \equiv [&\mathit{update}\_\langle \mathsf{int}^{2m+1}, \mathsf{D}^{2m-1}, \updownarrow \mathsf{P}\rangle, \mathit{write}\_\langle \mathsf{int}^{2m+1}, (\updownarrow \mathsf{P})^{2m+1}\rangle,\\ &\mathit{root}\_\langle \mathsf{int}^{2m+1}, (\updownarrow \mathsf{P})^{2m+1}\rangle, \mathit{empty}\_\langle \mathsf{int}, \updownarrow \mathsf{P}\rangle, \mathit{full}\_\langle\,\rangle, \mathit{emptyr}\_\langle\,\rangle,\\ &\mathit{done}\_\langle\,\rangle, \mathit{del}\_\updownarrow \mathsf{P}, \mathit{replace}\_\langle \mathsf{int}, \updownarrow \mathsf{P}\rangle, \mathit{remove}\_\updownarrow \mathsf{P}\,].\end{aligned}$$

Here we give the definition of a leaf; the definitions of NODE and ROOT are given in the Appendix. Since the high key of a leaf is a vital guide to operation-agents, it is duplicated so that even if it and the associated pointer are deleted, a copy of the key remains. Thus the definition of agent $\mathsf{LEAF}\,\langle p, \tilde{k}, \tilde{d}, q\rangle$ representing a leaf

named p storing keys $\tilde{k} = k_1 \cdots k_{j-1}$, high key k_j, pointers $\tilde{d} = d_2 \cdots d_{j-1}$ to database records and link q is

$$
\begin{aligned}
&\mathsf{LEAF}(p, \tilde{k}, \tilde{d}, q) \\
&\stackrel{\text{df}}{=} p(z).\mathsf{case}(z) \\
&\quad [search\_\langle k, r\rangle : \\
&\quad \mathsf{cond}(k > k_j \ \triangleright\ \bar{r}(link\_q).L, \\
&\qquad\quad k = k_h \ \triangleright\ \bar{r}(datum\_d_h).L, \\
&\qquad\quad k \notin \tilde{k} \ \triangleright\ \bar{r}(datum\_nil).L), \\
&\quad insert\_\langle k, d, r\rangle : \\
&\quad \mathsf{cond}(k > k_j \ \triangleright\ \bar{r}(link\_q).L, \\
&\qquad\quad k = k_h \ \triangleright\ \bar{r}(done\_\langle\,\rangle).L_h, \\
&\qquad\quad notfull \ \triangleright\ \bar{r}(done\_\langle\,\rangle).L', \\
&\qquad\quad full \ \triangleright\ (vp')\,\bar{r}(split\_\langle p', k'\rangle).(L_1 \mid L_2)), \\
&\quad delete\_\langle k, r\rangle : \\
&\quad \mathsf{cond}(k > k_j \ \triangleright\ \bar{r}(link\_q).L, \\
&\qquad\quad k \notin \tilde{k} \ \triangleright\ (done\_\langle\,\rangle).L, \\
&\qquad\quad notempty \ \triangleright\ \bar{r}(done\_\langle\,\rangle).L_k, \\
&\qquad\quad empty \ \triangleright\ \bar{r}(empty\_\langle\,\rangle).L_k), \\
&\quad compress\_\langle r\rangle : \\
&\quad (vu)\,\bar{r}(data\_\langle \tilde{k}, \tilde{d}, q, u\rangle).u(w).\mathsf{case}(w)[update\_\langle \tilde{k}', \tilde{d}', q'\rangle : \mathsf{LEAF}\,\langle p, \tilde{k}', \tilde{d}', q'\rangle, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad del\_q' \qquad\qquad : \mathsf{DELETED}\,\langle p, q'\rangle\,]],
\end{aligned}
$$

where the Boolean expressions *empty* and *notempty* are $j \leqslant m$ and $j > m$, respectively. In its quiescent state a leaf may accept, in addition to search and insert requests, requests for deletions and compressions. Search and insert requests are handled as before with the exception that the free variable h now ranges over $2 \cdots j-1$ and in the case where the leaf is split we have

1. if $k_{m+1} \leqslant k_h < k < k_{h+1}$ then

$$L_1 = \mathsf{LEAF}\,\langle p, k_1 \cdots k_{m+1} k_{m+1}, b_2 \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\,\langle p', k_{m+1} \cdots k_h\, k\, k_{h+1} \cdots k_{2m+1}, b_{m+2} \cdots b_h\, b\, b_{h+1} \cdots b_{2m}, q\rangle,$$

2. if $k_h < k < k_{h+1} \leqslant k_{m+1}$ then

$$L_1 = \mathsf{LEAF}\,\langle p, k_1 \cdots k_h\, k\, k_{h+1} \cdots k_{m+1} k_{m+1}, b_2 \cdots b_h\, b\, b_{h+1} \cdots b_{m+1}, p'\rangle$$

$$L_2 = \mathsf{LEAF}\,\langle p', k_{m+1} \cdots k_{2m+1}, b_{m+2} \cdots b_{2m}, q\rangle.$$

A delete request contains an integer k to be deleted and a name r via which to return the result. If k is greater than the high of the leaf, the link is returned. If k is not present in the leaf then $done\_\langle\,\rangle$ is returned. Otherwise the deletion is performed,

$$L_k = \mathsf{LEAF}\,\langle p, k_1 \cdots k_{h-1} k_{h+1} \cdots k_j, d_2 \cdots d_{h-1} d_{h+1} \cdots d_j, q\rangle,$$

where $k = k_h$. If as a result of the deletion the leaf has become less than half full, the value $empty\_\langle\ \rangle$ is returned; the recipient of this will activate a compressor process to rebalance the tree so that all nodes are at least half full. On receiving a compress request containing a name r, the leaf emits via r its contents and a fresh name u. It may then receive via u a tuple of the form $\langle \tilde{k}', \tilde{d}', q' \rangle$ tagged with label *upd*, or a single pointer q' tagged with label *del*. In the former case the leaf updates its contents to be the data received, in the latter it becomes empty; the agent DELETED $\langle p, q \rangle$ is defined and explained below.

We continue with the deletion algorithm of [16]. A deletion is effected by locating the appropriate leaf and then requesting removal of the key and the associated pointer. Thus execution of a deletion is somewhat similar to that of an insertion where no splitting occurs:

$$
\begin{aligned}
&D_0(d, get)\\
&\quad \stackrel{\text{df}}{=} !d(k, a_d).get(p).Delete\langle k, a_d, p, \langle\ \rangle\rangle\\
&Delete(k, a_d, p, \tilde{q})\\
&\quad \stackrel{\text{df}}{=} (vr)\ \bar{p}(search\_\langle k, r\rangle).r(y).\\
&\qquad \mathsf{cond}(\tilde{q} = \langle\ \rangle \triangleright \mathsf{case}(y)[link\_p' \quad : Delete\langle k, a_d, p', \langle p\rangle\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad nonlink\_p' : Delete\langle k, a_d, p', \langle p\rangle\rangle],\\
&\qquad\qquad \tilde{q} \neq \langle\ \rangle \triangleright \mathsf{case}(y)[link\_p' \quad : Delete\langle k, a_d, p', \tilde{q}\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad nonlink\_p' : Delete\langle k, a_d, p', p\tilde{q}\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad datum\_b' \ : Del\langle k, a_d, p, \tilde{q}\rangle])\\
&Del(k, a_d, p, \tilde{q})\\
&\quad \stackrel{\text{df}}{=} (vr)\ \bar{p}(delete\_\langle k, r\rangle).r(y).case(y)[link\_p' \quad : Del\langle k, a_d, p', \tilde{q}\rangle,\\
&\qquad\qquad\qquad\qquad\qquad\qquad done\_\langle\ \rangle \ : \overline{a_d}.\mathbf{0},\\
&\qquad\qquad\qquad\qquad\qquad\qquad empty\_\langle\ \rangle : \overline{a_d}.\bar{c}(p, k, \tilde{q}).\mathbf{0}].
\end{aligned}
$$

The agent D_0 may repeatedly generate deletion processes when supplied via name d with a key k to be deleted and a name a_d via which to signal completion of the operation. *Delete* follows a path through the tree, recording its starting point and the rightmost node visited at each level. When the appropriate leaf is found, it requests deletion of key k. If the deletion results in the leaf becoming less than half full, via the name c a compression process is activated to redistribute the leaf's data or delete it if it has become empty; this may lead to activation of other compression processes.

We continue to consider the compression algorithm of [16]. Rather than describing the entities involved as agents, we will discuss a certain defect of the algorithm and then present an improvement of it using agents in the Appendix.

Using the names $\tilde{q}$ received, the compressor of [16] first locates and locks the parent P of the leaf A to be compressed. It then locks and examines leaf A. If A is no longer less than half full, due to insertions and compressions having taken place since the compressor's creation, the compressor terminates. Similarly, if A is the rightmost child of P, the two cells are unlocked and the compressor terminates. Otherwise, the third cell to be locked is A's right neighbour, B. If P does not have a pointer to B then it must be that this is still to be inserted in P by an inserter in

its *Up* phase. In this case all three cells are unlocked and the compressor repeats this part of its activity, beginning by locking P. It is expected that in the meantime a pointer to B will have been added to P and the compressor will be able to proceed. If, on the other hand, P does have a pointer to B, then one of the following takes place:

1. If A and B have together no more than $2m$ pairs, the data of B are moved into A, leaf B is deleted, and the old high key of A and the pointer to B are deleted from P. Then the cells are unlocked. It is possible that due to the deletion of a pair from node P, it may become less than half full. If either of A and P is less than half full then further compression processes are initiated to rebalance the tree. This process may be repeated in several levels of the tree and may reach the root. As with insertion agent *Up*, it is possible that the path provided by the deletion process may become empty although a compression is required at a higher level of the tree. If this happens, the compression process queries the STORE to obtain the name of the leftmost node at the level above.

2. If A and B together have more than $2m$ pairs then pairs are moved from B to A so that each has at least m pairs. The high key of A is updated in P and the three cells are unlocked.

The compression algorithm may result in a leaf (or other cell) becoming empty. It is possible, however, that an operation-agent has a pointer to this leaf. Hence a leaf cannot simply be removed. Instead, when a leaf becomes empty its data is replaced by a pointer to the leaf where the search should continue: the leaf whither its data is moved. The agent DELETED $\langle p, q \rangle$ representing an empty cell with name p and storing pointer q is defined by

$$\begin{aligned}\mathsf{DELETED}(p, q) \stackrel{\text{df}}{=} p(z).\mathsf{cond}[\, & search\_\langle k, r\rangle : \bar{r}(link\_q).\mathsf{DELETED}\ \langle p, q\rangle, \\ & \cdots \\ & compress\_\langle r\rangle : \bar{r}(link\_q).\mathsf{DELETED}\ \langle p, q\rangle\,].\end{aligned}$$

An empty leaf (or other node) responds to any request by returning the stored pointer.

The algorithm has the following defect. Suppose a compressor and a searcher are executing concurrently. Suppose the compressor is at a leaf A and that case (2) above applies so that data of A's right neighbour B is to be moved to A. Suppose the searcher has progressed so that it is about to examine B which contains its target key k. Suppose further that k is among the data to be moved. The search will then continue at B and fail erroneously. This problem was noted in [16] and a solution was proposed in which processes are aborted and restarted. This involves storing the low key of each node explicitly and modifying the search phase of *all* of the operations so that during the search in a node for a key k, it is checked whether k is greater than the node's low key. If it is, the operation may proceed; otherwise it is aborted and must be restarted at a higher level.

The compression algorithm we propose is a variant of the algorithm above which avoids the problem described. Its design was guided by the proof of correctness in

Section 4. The intention was to ensure that, like *Up*, the compressor affects neither the contents of the tree nor the accessibility of nodes. Hence its actions do not change the $\simeq$-state of the system.

As mentioned before, an invariant maintained by the search and insertion algorithms crucial to their correctness is that the minimum key of a node is not altered during the node's lifetime. This and the fact that data is moved only from left to right ensures that operations can always be completed successfully. In fact a weaker property suffices: that the minimum of a node is never increased. Note that an execution of the compression algorithm of [16] may violate this property.

The new algorithm maintains this invariant during the compression process: when a leaf A is half empty the compressor locates and locks its parent P. If P has a pointer to A's left neighbour B, the compressor locks B and then A. (Note that this is in contrast to the algorithm of [16] where A's *right* neighbour is locked instead.) If A is no longer half empty, the compressor releases the three cells and terminates. If the link pointer of B does not point to A then all three cells are unlocked and the compressor repeats this part of its activity, beginning by locking P. Otherwise, one of the following takes place: if A and B have together no more than $2m$ pairs, the data of A is moved into B, leaf A is deleted, and the old high key of B and the pointer of A are deleted from P. This deletion may cause the activation of compression processes in higher levels of the tree, as described in (1) above. Otherwise, if A and B have together more than $2m$ pairs then the compressor performs (2) above.

Thus either A is deleted and all its data is moved into B, or data is moved from B to A. In the first case the low key of B remains the same, and as A is deleted it is made to point to node B. In the second case A's lowest key decreases and data moves from left to right.

If, however, A is the leftmost child of a node then this procedure cannot be applied, A having no left neighbour in the subtree rooted at P. If A is the only child of P then the nodes are unlocked and the compressor repeats this part of the activity beginning by locking P. It is expected that in the meantime node P will have been compressed. Otherwise, A's right neighbour C is visited. If A and C together have fewer than $2m+1$ keys, then C is deleted and its data is moved to A. Otherwise, the compressor releases the three nodes and repeats this part of its activity, beginning by locking P. The algorithm does not move data from C to A: this may cause failure of a process that subsequently tries to read C expecting to find information that has been moved to A. Although this last scenario may result in the compressor locking and unlocking the three nodes several times and delay the progress of other processes, we may expect that it is likely to arise infrequently due to the normal movement of data from left to right.

Hence the new algorithm maintains the invariants necessary to guarantee its correctness and that of the other operations, and in contrast to [16], none of the other operations must be changed, and the abortion and restarting of operations is avoided. Moreover, the new algorithm does not require the addition of the low key to the nodes of the data structure and is at least as efficient as the original algorithm. The process-calculus description of the compression algorithm, and of the modified NODE, ROOT, and STORE agents, are given in the Appendix.

Let P_0^+ be the system consisting of the initial data structure and the operations S_0, I_0, D_0 and C_0 (the generator of compressor agents). Let $B^+\langle s, i, d, f, \sigma, \sigma^c, \iota, \iota^c, \delta, \delta^c\rangle$ be defined as follows, where d is the name via which deletions may be initiated, δ (the *deletions*) is a set of pairs consisting of a key k to be deleted and a name a_d via which to signal completion, and a set δ^c (the *completed deletions*) of names a_d whose keys have been deleted but which have not been used to signal this:

$$
\begin{aligned}
B^+ \stackrel{\text{df}}{=} & s(k, a).B^+\langle ..., \sigma \cup \{\langle k, a_s\rangle\}, ...\rangle \\
& + i(k, b, a_i).B^+\langle ..., \iota \cup \{\langle k, b, a_i\rangle\}, ...\rangle \\
& + d(k, a_d).B^+\langle ..., \delta \cup \{\langle k, a_d\rangle\}, ...\rangle \\
& + \Sigma_{\langle k, a_s\rangle \in \sigma}\, \tau.B^+\langle ..., \sigma - \{\langle k, a_s\rangle\}, \sigma^c \cup \{\langle a_s, f(k)\rangle\}, ...\rangle \\
& + \Sigma_{\langle k, b, a_i\rangle \in \iota}\, \tau.B^+\langle ..., f[b/k], ..., \iota - \{\langle k, p, a_i\rangle\}, \iota^c \cup \{a_i\}, ...\rangle \\
& + \Sigma_{\langle k, a_d\rangle \in \delta}\, \tau.B^+\langle ..., f[nil/k], ..., \delta - \{\langle k, a_d\rangle\}, \delta^c \cup \{\langle a_d\rangle\}\rangle \\
& + \Sigma_{\langle a_s, b\rangle \in \sigma^c}\, \overline{a_s}\langle b\rangle.B^+\langle ..., \sigma^c - \{\langle a_s, b\rangle\}, ...\rangle \\
& + \Sigma_{a_i \in \iota^c}\, \overline{a_i}.B^+\langle ..., \iota^c - \{\langle a_i\rangle\}, ...\rangle \\
& + \Sigma_{a_d \in \delta^c}\, \overline{a_d}.B^+\langle ..., \delta^c - \{\langle a_d\rangle\}\rangle.
\end{aligned}
$$

Let $B_0^+ = B^+\langle i, s, d, \lambda k.nil, \varepsilon, \varepsilon, \varepsilon, \varepsilon, \varepsilon\rangle$. The result asserting the correctness of the operations is the following:

THEOREM 5.1. $P_0^+ \simeq_{\mathsf{A}} B_0^+$.

Proof. The proof is an extension of that of Theorem 4.1. It involves a detailed analysis to show that a deleter does indeed perform a single decisive τ-action, and that a compressor does not alter the $\simeq$-state of the system. The proof is omitted. Note that branching bisimilarity does not distinguish processes that differ only in that one can diverge while the other cannot. ∎

6. CONCLUSION

This paper has presented an analysis of concurrent operations on B^{link}-trees using the π-calculus. The process calculus made possible a natural and direct description of both data and algorithms and of change in the system's structure as it evolves. Moreover, the general theory of process equivalence was used to assert the correctness of the algorithms in terms of the observable behaviour of the system. The calculus helped in clarifying certain details of the algorithms, and in guiding us to consider some cases absent from the pseudo-code descriptions in [6]. Moreover, the calculus was invaluable in structuring the proof of correctness. We believe the proof offers increased confidence over previous proofs, and gives useful insight into why the operations are correct. The insight may be helpful in considering design alternatives. It may also provide a guide for analysis of operations on other search structures. Indeed, insight gained from the proof was vital in discovering the improvement to the deletion and compression algorithms of [16].

The paper has also presented some general results on partially confluent processes and client-server systems. The results, which extend work in [7], make precise that in reasoning about compositions of partially confluent agents, it is often sufficient to examine in detail only a part of the composite behaviour. The main result establishes that under certain conditions, a system composed of a client Q and a server A that interact in a question–answer fashion with possibly many questions outstanding at any moment, is behaviourally equivalent to a system composed from Q and a sequential part of A. The conditions on Q and on A are fairly mild. In particular, on accepting a question from Q, A may perform at most one state-changing internal action before producing an answer. Many systems, including algorithms proposed for dynamic search structures, appear to satisfy the conditions; thus the result may be helpful in proving their correctness.

The result was applied twice in proving the correctness of the B^{link}-tree operations. This made it possible to focus on the part of the system in which at most one operation is outstanding at any moment. This part is essentially a sequential system, and it was therefore much easier to understand and analyse than the full system. The main challenge was to establish that the hypotheses of the main theorem are met. In the case of the B^{link}-tree this could be done by fairly straightforward means. We believe it would be worthwhile to develop other techniques to help with this task, for instance based on type systems, and to seek other results in a similar vein.

APPENDIX

In this appendix we present the process-calculus descriptions of the compress operation and the modified NODE, ROOT, and STORE agents.

The definition of a nonroot, nonleaf node named p storing keys $\tilde{k} = k_1 \cdots k_j$ and pointers $\tilde{p} = p_1 \cdots p_j$ to nodes is as follows:

$$
\begin{aligned}
&\mathsf{NODE}(p, \tilde{k}, \tilde{p}) \\
&\quad \stackrel{\text{df}}{=} p(z).\mathsf{case}(z) \\
&\qquad [search\_\langle k, r\rangle: \\
&\qquad \mathsf{cond}(k > k_j \quad\quad\quad\quad \triangleright\ \bar{r}(link\_p_j).N, \\
&\qquad\qquad k_{h+1} > k \geqslant k_h \ \triangleright\ \bar{r}(nonlink\_p_h).N), \\
&\qquad add\_\langle k, q, r\rangle: \\
&\qquad \mathsf{cond}(k > k_j \ \triangleright\ \bar{r}(link\_p_j).N, \\
&\qquad\qquad k = k_h \ \triangleright\ \bar{r}(done\_\langle\,\rangle).N_h, \\
&\qquad\qquad notfull \triangleright\ \bar{r}(done\_\langle\,\rangle).N', \\
&\qquad\qquad full \quad\ \triangleright\ (vp')\,\bar{r}(split\_\langle p', k_{m+1}\rangle).(N_1 \mid N_2)), \\
&\qquad findp\_\langle k, q, r\rangle: \\
&\qquad \mathsf{cond}(parent \triangleright\ \bar{r}(retry\_\langle\,\rangle).N, \\
&\qquad\qquad k > k_j \ \triangleright\ \bar{r}(link\_p_j).N, \\
&\qquad\qquad q \in \tilde{p} \quad \triangleright\ (vu)\,\mathsf{cond}(q = p_1 \ \triangleright\ \bar{r}(right\_\langle p_2, u\rangle).\mathsf{NODE}^L\langle u\rangle, \\
&\qquad\qquad\qquad\qquad\qquad q = p_h \ \triangleright\ \bar{r}(left\_\langle p_{h-1}, u\rangle).\mathsf{NODE}^L\langle u\rangle)), \\
&\qquad compress\_\langle r\rangle: \\
&\qquad (vu)\,\bar{r}(contents\_\langle \tilde{k}, \tilde{p}, u\rangle).u(w).\mathsf{case}(w)[root\_\langle \tilde{k}', \tilde{p}'\rangle \ : \mathsf{ROOT}\,\langle p, \tilde{k}', \tilde{p}'\rangle, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad write\_\langle \tilde{k}', \tilde{p}'\rangle : \mathsf{NODE}\,\langle p, \tilde{k}', \tilde{p}'\rangle, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad del\_q \quad\quad\quad\quad : \mathsf{DELETED}\,\langle p, q\rangle\,]\,]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{NODE}^{L}(u)\\
&\quad \stackrel{\text{df}}{=} u(y).\mathsf{case}(y)\\
&\qquad [\mathit{done}\_\langle\,\rangle \quad : N,\\
&\qquad\ \mathit{replace}\_\langle k, q\rangle : N_k,\\
&\qquad\ \mathit{remove}\_q \quad : \mathsf{cond}(\mathit{empty} \quad \triangleright\ \bar{u}(\mathit{empty}\_\langle k_1, p\rangle).N_q,\\
&\qquad\qquad\qquad\qquad \mathit{notempty} \triangleright\ \bar{u}(\mathit{full}\_\langle\,\rangle).N_q)],
\end{aligned}
$$

where the Boolean expression *parent* is true when $(k \leqslant k_j \wedge q \notin \tilde{p}) \vee j = 2$. Thus, in its quiescent state a node may accept, in addition to search and add requests, requests for searches for the parent of a node and requests for compressions. The former contains the low key k of the node whose parent is to be found, its name p and a name r via which the return of the search should be returned. If k belongs to the range of the node but the pointer is not one of the node's pointers, or the node has only one child, that is if *parent* is true, then the node returns via r an empty tuple tagged with the label *retry*. The recipient of this message, the agent responsible for initiating the search, will repeat this part of the activity at a later point. It is expected that in the meantime, either pointer q will have been added to the node (if $q \notin \tilde{p}$), or the node will have been compressed (if $j = 2$) and thus the search for the parent of q will proceed. On the other hand, if k is larger than the high key of the node, the link is returned. Otherwise, one of the following takes place: if p is the leftmost child of the node, p_2, the pointer to its right neighbour tagged with label *right* is returned via r, otherwise a pointer to its left neighbour tagged with label *left* is returned. In each case the pointer is accompanied by a fresh name u of type $\updownarrow \mathsf{U}$ and the node assumes state $\mathsf{NODE}^{L}\langle u\rangle$. In this state the node may accept a message along name u. If the message $\mathit{done}\_\langle\,\rangle$ is received then the node enters the quiescent state $N = \mathsf{NODE}\langle p, \tilde{k}, \tilde{p}\rangle$. If the pair $\mathit{replace}\_\langle k, q\rangle$ is received then the key associated with pointer q is updated to k;

$$N_k = \mathsf{NODE}\langle p, k_1 \cdots k_{h-1} k k_{h+1} \cdots k_j, \tilde{p}\rangle,$$

where $q = p_h$. Finally, if message remove_q is received, pointer q and its associated key are removed fro the node

$$N_q = \mathsf{NODE}\langle p, k_1 \cdots k_{h-1} k_{h+1} \cdots k_j, p_1 \cdots p_{h-1} p_{h+1} \cdots p_j\rangle,$$

where $q = p_h$. Via u the message $\mathit{empty}\_\langle k_1, p\rangle$ is returned, if the node has become less than half full, and the message $\mathit{full}\_\langle\,\rangle$, otherwise.

Finally, in its quiescent state a node may receive compress requests, containing a name r via which the node emits its contents and a fresh name, u. It then receives a message via u which may have one of three forms: it may contain a tuple of the form $\langle \tilde{k}', \tilde{p}'\rangle$ tagged with label *root*, or with label *write*, or a pointer tagged with label *del*. In the first two cases the node updates its contents to the data received and if the message was tagged with the *root* label it assumes root status. In the last case the node is deleted.

The agent $\mathsf{ROOT}\langle p, \tilde{k}, \tilde{p}, put\rangle$ representing a root named p storing keys $\tilde{k} = k_1 \cdots k_j$ and pointer $\tilde{p} = p_1 \cdots p_j$ to nodes is defined by

$$
\begin{aligned}
&\mathsf{ROOT}(p, \tilde{k}, \tilde{p}, put)\\
&\stackrel{\mathrm{df}}{=} p(z).\mathsf{case}(z)\\
&\quad [search\_\langle k, r\rangle :\\
&\quad\ \mathsf{cond}(k_{h+1} \geqslant k > k_h \rhd \bar{r}(nonlink\_p_h).R),\\
&\quad\ add\_\langle k, q, r\rangle :\\
&\quad\ \mathsf{cond}(k = k_h \rhd \bar{r}(done\_\langle\ \rangle).R_h,\\
&\qquad\qquad notfull \rhd \bar{r}(done\_\langle\ \rangle).R',\\
&\qquad\qquad full \quad \rhd (vp_0, p')\ put(p_0).\bar{r}(done\_\langle\ \rangle).(\mathsf{NEWROOT} \mid N_1 \mid N_2)),\\
&\quad\ findp\_\langle k, q, r\rangle :\\
&\quad\ \mathsf{cond}(parent \rhd \bar{r}(retry\_\langle\ \rangle).R,\\
&\qquad\qquad q \in \tilde{p} \quad \rhd (vu)\ \mathsf{cond}(q = p_1 \rhd \bar{r}(right\_\langle p_2, u\rangle).\mathsf{ROOT}^L,\\
&\qquad\qquad\qquad\qquad\qquad q = p_h \rhd \bar{r}(left\_\langle p_{h-1}, u\rangle).\mathsf{ROOT}^L))]
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{ROOT}^L\\
&\stackrel{\mathrm{df}}{=} u(z).\mathsf{case}(z)\\
&\qquad [done\_\langle\ \rangle \qquad : R,\\
&\qquad\ replace\_\langle k, q\rangle : R_k,\\
&\qquad\ remove\_q \qquad : \mathsf{cond}(j = 3 \rhd \bar{u}(emptyr\_\langle\ \rangle).u(w).\mathsf{case}(w)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad [done\_\langle\ \rangle : R_q,\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ del\_q : \mathsf{DELETED}\langle p, q\rangle]\\
&\qquad\qquad\qquad\qquad\qquad\qquad j > 3 \rhd \bar{u}(full\_\langle\ \rangle).R_q)].
\end{aligned}
$$

Search and add requests are handled as before. The response to *findp*-tagged requests is similar to that of a node

$$R_k = \mathsf{ROOT}\langle p, k_1 \cdots k_{h-1} k k_{h+1} \cdots k_j, \tilde{p}\rangle,$$

where $q = p_h$. In the final alternative, following the receipt of message $remove\_q$, the root deletes the pointer q from its contents

$$R_q = \mathsf{ROOT}\langle p, k_1 \cdots k_{h-1} k_{h+1} \cdots k_j, p_1 \cdots p_{h-1} p_{h+1} \cdots p_j\rangle,$$

where $q = p_h$, and if the resulting root has a single remaining child, that is if $j = 3$, it signals on name u that it has become empty by sending the message $emptyr\_\langle\ \rangle$. It then accepts a message on u and enters state R_q or becomes a deleted node, $\mathsf{DELETED}\langle p, q\rangle$, where q is the pointer received from the agent responsible for the deletion.

We now proceed to the formal definition of the compression algorithm in the process calculus. First note that since compressions may result in the deletion of nodes including the root, it is necessary that the STORE agent is notified of such deletions and is kept updated regarding the identity of the current root. Thus agent

$\mathsf{STORE}\langle \tilde{p}, get, put, next, rem\rangle$, where $\tilde{p} = p_1 \cdots p_j$ are the names of the current and previous roots currently present in the data structure in order of creation, is defined by

$$\begin{aligned}\mathsf{STORE}(\tilde{p}, get, put, next, rem) \stackrel{\text{df}}{=} {} & \overline{get}(p_j).S \\ & + put(p).S' \\ & + next(p, n).\mathsf{cond}(p = p_h \rhd \bar{n}(p_{h+1}).S) \\ & + rem(p).\mathsf{cond}(p = p_h \rhd S_h),\end{aligned}$$

where $S = \mathsf{STORE}(\tilde{p}, get, put, next, rem)$, $S' = \mathsf{STORE}(\tilde{p}p, get, put, next, rem)$, and $S_h = \mathsf{STORE}(p_1 \cdots p_{h-1}, get, put, next, rem)$. Thus the store may receive via name *rem* the name of a root, q, that has been deleted. In response to such a message it removes pointer q and all pointers that have named the root after q.

When the compression agent $C_0\langle c, rem\rangle$ below is supplied with the pointer to the half-empty leaf and the path recorded by the deletion process, it activates a compressor responsible for redistributing the leaf's data or deleting it if it has become empty:

$$\begin{aligned}& C_0(c, rem) \\ & \quad \stackrel{\text{df}}{=} !\, c(p, k, \tilde{p}).FindP_L\langle p_1, p, k, \tilde{p}\rangle\end{aligned}$$

$$\begin{aligned}& FindP_L(q, p, k, \tilde{p}) \\ & \quad \stackrel{\text{df}}{=} (vr)\, \bar{q}(findp\_\langle k, p, r\rangle).r(y). \\ & \qquad \mathsf{case}(y)[retry\_\langle\,\rangle : FindP_L\langle q, p, k, \tilde{q}\rangle, \\ & \qquad\qquad link\_q' : FindP_L\langle q', p, k, \tilde{p}\rangle, \\ & \qquad\qquad left\_\langle q', u\rangle : CompL_L\langle q', p, k, \tilde{p}, u\rangle, \\ & \qquad\qquad right\_\langle q', u\rangle : CompR_L\langle p, q', k, \tilde{p}, u\rangle]\end{aligned}$$

$$\begin{aligned}& CompL_L(p, q, k, \tilde{p}, u) \\ & \quad \stackrel{\text{df}}{=} (vr, s_1, s_2, s_3)\, \bar{p}(compress\_r).r(data\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1, u_1\rangle). \\ & \qquad \mathsf{cond}(q_1 \neq q \rhd \overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1\rangle).\bar{u}(done\_\langle\,\rangle).FindP_L\langle p_1, q, k, \tilde{p}\rangle, \\ & \qquad\quad q_1 = q \rhd (vr)\, \bar{q}(compress\_r).r(data\_\langle \widetilde{k_2}, \widetilde{d_2}, q_2, u_2\rangle). \\ & \qquad\qquad \mathsf{cond}(j_1 > m \rhd \overline{s_1}.\mathbf{0}, \\ & \qquad\qquad\quad j_1 + j_2 > 2m + 2 \rhd \overline{s_2}.\mathbf{0}, \\ & \qquad\qquad\quad j_1 + j_2 \leqslant 2m + 2 \rhd \overline{s_3}.\mathbf{0})) \\ & \qquad | \, s_1.\overline{u_2}(update\_\langle \widetilde{k_2}, \widetilde{d_2}, q_2\rangle).\overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1\rangle).\bar{u}(done\_\langle\,\rangle).\mathbf{0} \\ & \qquad | \, s_2.\overline{u_2}(update\_\langle \widetilde{k_2}', \widetilde{d_2}', q_2\rangle).\overline{u_1}(update\_\langle \widetilde{k_1}', \widetilde{d_1}', q_1\rangle).\bar{u}(replace\_\langle k', q\rangle).\mathbf{0} \\ & \qquad | \, s_3.\overline{u_1}(update\_\langle \widetilde{k_1}'', \widetilde{d_1}'', q_2\rangle).\overline{u_2}(del\_p).\bar{u}(remove\_q).u(y). \\ & \qquad\quad \mathsf{case}(y)[full\_\langle\,\rangle : \mathbf{0}, \\ & \qquad\qquad\quad emptyr\_\langle\,\rangle : \bar{u}(done\_\langle\,\rangle).\mathbf{0}, \\ & \qquad\qquad\quad empty\_\langle p', k'\rangle : C_N\langle p', k', \tilde{p}\rangle]\end{aligned}$$

$$
\begin{aligned}
&CompR_L(p, q, k, \tilde{p}, u)\\
&\quad \stackrel{\text{df}}{=} (vr, s_1, s_2)\ \bar{p}(compress\_r).r(data\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1, u_1\rangle).\\
&\qquad \mathsf{cond}(q_1 \neq q \rhd \overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1\rangle).\bar{u}(done\_\langle\ \rangle).\mathbf{0},\\
&\qquad\qquad j_1 > m \rhd \overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1\rangle).\bar{u}(done\_\langle\ \rangle).\mathbf{0},\\
&\qquad\qquad j_1 \leqslant m \rhd (vr)\ \bar{q}(compress\_r).r(data\_\langle \widetilde{k_2}, \widetilde{d_2}, q_2, u_2\rangle).\\
&\qquad\qquad\qquad \mathsf{cond}(j_1 + j_2 > 2m+2 \rhd \overline{s_1}.\mathbf{0},\\
&\qquad\qquad\qquad\qquad j_1 + j_2 \leqslant 2m+2 \rhd \overline{s_2}.\mathbf{0}))\\
&\qquad \mid s_1.\overline{u_2}(update\_\langle \widetilde{k_2}, \widetilde{d_2}, q_2\rangle).\overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{p_1}, q_1\rangle).\bar{u}(done\_\langle\ \rangle).\mathbf{0}\\
&\qquad \mid s_2.\overline{u_1}(update\_\langle \widetilde{k_1}'', \widetilde{d_1}'', q_2\rangle).\overline{u_2}(del\_p).\bar{u}(remove\_q).u(y).\\
&\qquad\quad \mathsf{case}(y)[\,full\_\langle\ \rangle \quad\quad : \mathbf{0},\\
&\qquad\qquad\quad emptyr\_\langle\ \rangle \quad\quad : \bar{u}(done\_\langle\ \rangle).\mathbf{0},\\
&\qquad\qquad\quad empty\_\langle p', k'\rangle : C_N\langle p', k', \tilde{p}\rangle\,].
\end{aligned}
$$

Thus, using the path received, the compressor first searches for the parent of the leaf A to be compressed at the level above, until a node synchronizes with it by performing one of the actions $\bar{r}(left\_\langle q', u\rangle)$, $\bar{r}(right\_\langle q', u\rangle)$, where q' is the name of a neighbour B of A. Receipt of the first message results in activation of agent $CompL_L$, whereas receipt of the second message triggers execution of agent $CompR_L$. Agent $CompL_L$ begins by reading the leftmost of the two leaves, B, and establishing whether it points to A. If not, the leaf and its parent are unlocked via actions $\overline{u_1}(update\_\langle \widetilde{k_1}, \widetilde{d_1}, q_1\rangle)$ and $\bar{u}(done\_\langle\ \rangle)$, otherwise the compressor proceeds to read A. Three cases exist:

1. If A is no longer less than half empty, that is if $j_2 > m$, all nodes are unlocked.

2. If the two leaves together have more than $2m+2$ pairs then pairs are shifted from B to A: if $\widetilde{k_1} = k_{11} \cdots k_{1j_1}$, $\widetilde{d_1} = d_{12} \cdots d_{1(j_1-1)}$ and $\widetilde{k_2} = k_{21} \cdots k_{2j_2}$, $\widetilde{d_2} = d_{22} \cdots d_{2(j_2-1)}$, letting $n = \lceil (j_1 + j_2)/2 \rceil$

$$
\begin{aligned}
\widetilde{k_1}' &= k_{11} \cdots k_{1n} k_{1n}\\
\widetilde{k_2}' &= k_{1n} \cdots k_{1(j_1-1)} k_{22} \cdots k_{2j_2}\\
\widetilde{d_1}' &= d_{11} \cdots d_{1n}\\
\widetilde{d_2}' &= d_{1(n+1)} \cdots d_{1(j_1-1)} \widetilde{d_2}.
\end{aligned}
$$

Moreover, the parent of the nodes is informed that the low key of A has been altered via return of the message $replace\_\langle k', q\rangle$, where $k' = k_{1n}$ is the new low key of A.

3. If the two leaves together have fewer than $2m+2$ pairs then the contents of A are moved to B:

$$
\begin{aligned}
\widetilde{k_1}'' &= k_{11} \cdots k_{1(j_1-1)} k_{22} \cdots k_{2j_2}\\
\widetilde{d_1}'' &= \widetilde{d_1} \widetilde{d_2}
\end{aligned}
$$

Furthermore, the high key of B becomes the high key of A, and A becomes a deleted node pointing to B. This is achieved by the sending of message del_p. Note that the parent of the node is also informed that node A has been deleted by being sent the message $\langle q, r \rangle$ suitably tagged. The parent is then responsible for deleting the pointer q to leaf A and for informing the compressor via name r whether it has become less than half empty. If it has indeed become empty, and it is not the root, then the compressor activates another compressor agent to compress the node. The definition of agent C_N is given below.

Finally, agent $CompR_L$ behaves similarly to $CompL_L$. The main difference is that $CompR_L$ begins by locking A, being now the leftmost of the two leaves, and only then does it proceed to lock B. Furthermore, if the two leaves together have more than $2m$ pairs then no data is moved from B to A (as this would increase the low key of B) and instead all nodes are unlocked.

Whenever the compression agent C_N, defined below, is supplied with a tuple $\langle p, k, \tilde{p}, rem \rangle$ is undertakes the compression of the node A with name p and low key k. Note that here $\tilde{p}$ is a path from the root (or a node which was the root) to a node on the same level and to the left of A. The behaviour of C_N is similar to C_0 with the distinction that it deals with internal nodes rather than leaves:

$$C_N(p, k, \langle q \rangle, rem) \stackrel{\text{df}}{=} (vn)\, \overline{next}(\langle q, n \rangle).n(q').FindP_N\langle q', p, k, \langle q' \rangle\rangle$$

$$C_N(p, k, \tilde{p}, rem) \stackrel{\text{df}}{=} FindP_N\langle p_2, p, k, p_2 \cdots p_m \rangle.$$

If the path contains only one element then the compressor queries the store to obtain the pointer that named the root after q and proceeds to find the parent of A. If the path contains two or more elements, the compressor immediately undertakes that search. Note that p_2 is a pointer to a node one level higher than A.

$$\begin{aligned}
&FindP_N(q, p, k, \tilde{p}) \\
&\quad \stackrel{\text{df}}{=} (vr)\, \bar{q}(findp_\langle k, p, r \rangle).r(z). \\
&\qquad \mathsf{case}(z)[retry_\langle\,\rangle \;\; : FindP_N\langle q, p, k, \tilde{p} \rangle, \\
&\qquad\qquad link_q' \;\; : FindP_N\langle q', p, k, \tilde{p} \rangle, \\
&\qquad\qquad left_\langle q', u \rangle \;\; : CompL_N\langle q', p, k, \tilde{p}, u \rangle, \\
&\qquad\qquad right_\langle q', u \rangle : CompR_N\langle p, q', k, \tilde{p}, u \rangle]
\end{aligned}$$

$$\begin{aligned}
&CompL_N(p, q, k, \tilde{p}, u) \\
&\quad \stackrel{\text{df}}{=} (vr, s_1, s_2, s_3)\, \bar{p}(compress_r).r(contents_\langle \widetilde{p_1}, \widetilde{k_1}, u_1 \rangle). \\
&\qquad \mathsf{cond}(p_{1_{j_1}} \neq q \rhd \overline{u_1}(write_\langle \widetilde{k_1}, \widetilde{p_1} \rangle).\bar{u}(done_\langle\,\rangle).FindP_N\langle p_1, q, k, \tilde{p} \rangle, \\
&\qquad\qquad p_{1_{j_1}} = q \rhd (vr)\, \bar{q}(compress_r).r(\widetilde{p_2}, \widetilde{k_2}, u_2). \\
&\qquad\qquad\qquad \mathsf{cond}(j_2 > m \qquad\quad \rhd \overline{s_1}).\mathbf{0}, \\
&\qquad\qquad\qquad\qquad j_1 + j_2 > 2m + 2 \rhd \overline{s_2}.\mathbf{0}, \\
&\qquad\qquad\qquad\qquad j_1 + j_2 \leqslant 2m + 2 \rhd \overline{s_3}.\mathbf{0}))
\end{aligned}$$

$$
\begin{aligned}
&\mid s_1.\overline{u_2}(write\_\langle \widetilde{k}_2, \widetilde{p}_2\rangle).\overline{u_1}(write\_\langle \widetilde{k}_1, \widetilde{p}_1\rangle).\bar{u}(done\_\langle\,\rangle).\mathbf{0}\\
&\mid s_2.\overline{u_2}(write\_\langle \widetilde{k}_2', \widetilde{p}_2'\rangle).\overline{u_1}(write\_\langle \widetilde{k}_1', \widetilde{p}_1'\rangle).\bar{u}(replace\_\langle k', q\rangle).\mathbf{0}\\
&\mid s_3.\overline{u_2}(del\_p).\bar{u}(remove\_q).u(w).\\
&\quad \mathsf{case}(w)[\,full\_\langle\,\rangle \;\; : \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\mathbf{0},\\
&\qquad\qquad empty\_\langle k', p'\rangle : \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).C_N\langle p', k', \tilde{p}\rangle,\\
&\qquad\qquad emptyr\_\langle\,\rangle \;\; : \mathsf{cond}(p_{2j_2} \neq nil \rhd \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\\
&\qquad\qquad\qquad\qquad\qquad\qquad \bar{u}(done\_\langle\,\rangle).\mathbf{0},\\
&\qquad\qquad\qquad\qquad\quad p_{2j_2} = nil \rhd \overline{u_1}(root\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\\
&\qquad\qquad\qquad\qquad\qquad\qquad \bar{u}(del\_p).\overline{rem}\,\langle p_1\rangle.\mathbf{0})]
\end{aligned}
$$

$$
\begin{aligned}
&CompR_N(p, q, k, \tilde{p}, u)\\
&\quad \stackrel{\text{df}}{=} (vr, s_1, s_2)\ \bar{p}(compress\_r).r(contents\_\langle \widetilde{p}_1, \widetilde{k}_1, u_1\rangle).\\
&\qquad \mathsf{cond}(p_{j_1} \neq q \rhd \overline{u_1}(write\_\langle \widetilde{k}_1, \widetilde{p}_1\rangle).\bar{u}(done\_\langle\,\rangle).\mathbf{0},\\
&\qquad\qquad j_1 > m \;\rhd \overline{u_1}(write\_\langle \widetilde{k}_1, \widetilde{p}_1\rangle).\bar{u}(done\_\langle\,\rangle).\mathbf{0},\\
&\qquad\qquad j_1 \leqslant m \;\rhd (vr)\,\bar{q}(compress\_r).r(contents\_\langle \widetilde{p}_2, \widetilde{k}_2, u_2\rangle).\\
&\qquad\qquad\qquad \mathsf{cond}(j_1 + j_2 > 2m + 2 \rhd \overline{s_1}.\mathbf{0},\\
&\qquad\qquad\qquad\qquad j_1 + j_2 \leqslant 2m + 2 \rhd \overline{s_2}.\mathbf{0}))\\
&\qquad \mid s_1.\overline{u_2}(write\_\langle \widetilde{k}_2, \widetilde{p}_2\rangle).\overline{u_1}(write\_\langle \widetilde{k}_1, \widetilde{p}_1\rangle).\bar{u}(done\_\langle\,\rangle).\mathbf{0}\\
&\qquad \mid s_2.\overline{u_2}(del\_p).\bar{u}(remove\_q).u(w).\\
&\qquad\quad \mathsf{case}(w)[\,full\_\langle\,\rangle \;\; : \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\mathbf{0},\\
&\qquad\qquad\quad empty\_\langle k', p'\rangle : \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).C_N\langle p', k', \tilde{p}\rangle,\\
&\qquad\qquad\quad emptyr\_\langle\,\rangle \;\; : \mathsf{cond}(p_{2j_2} \neq nil \rhd \overline{u_1}(write\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\\
&\qquad\qquad\qquad\qquad\qquad\qquad \bar{u}(done\_\langle\,\rangle).\mathbf{0},\\
&\qquad\qquad\qquad\qquad\quad p_{2j_2} = nil \rhd \overline{u_1}(root\_\langle \widetilde{k}_1'', \widetilde{p}_1''\rangle).\\
&\qquad\qquad\qquad\qquad\qquad\qquad \bar{u}(del\_p).\overline{rem}\langle p_1\rangle.\mathbf{0})]
\end{aligned}
$$

Agent $FindP_N$ behaves similarly to $FindP_L$. The main difference concerns the last alternatives of agents $CompL_N$ and $CompR_N$. In particular, if the compression results in the root becoming empty, which corresponds to the state where the root contains a single pair, and if additionally its child has no neighbours, then the root is deleted and its child becomes the root. In such a case, the compressor emits via name *rem* the name of the old root, p_1. This action synchronizes with agent STORE which, in response to the message, removes pointer p_1 and all pointers that have named the root after p_1, if any.

ACKNOWLEDGMENTS

We are grateful to the (anonymous) referees for helpful comments and suggestions.

REFERENCES

1. R. Bayer and E. McCreight, Organisation and maintenance of large ordered indexes, *Acta Inform.* **1** (1972), 173–189.

2. R. Bayer and M. Schkolnick, Concurrency of operations on B-trees, *Acta Inform.* **9** (1977), 1–21.
3. C. Ellis, Concurrency in linear hashing, *ACM Trans. Database Systems* **12** (1987), 195–217.
4. R. van Glabbeek and P. Weijland, Branching time and abstraction in bisimulation semantics, *in* "Information Processing '89," pp. 613–618, 1989.
5. Y. S. Kwong and D. Wood, A new method for concurrency in B-trees, *IEEE Trans. Software Eng.* **8** (1982), 211–222.
6. P. Lehman and S. B. Yao, Efficient locking for concurrent operations on B-trees, *AMS Trans. Database Systems* **6** (1981), 650–670.
7. X. Liu and D. Walker, Partial confluence of processes ans systems of objects, *Theoret. Comput. Sci.* **206** (1998), 127–162.
8. N. Lynch, M. Merritt, W. Weihl, and A. Fekete, "Atomic Transactions," Morgan Kaufmann, San Mateo, CA, 1994.
9. R. Miller and L. Snyder, Multiple access to B-trees, *in* "Information Science and Systems, Baltimore," 1978.
10. R. Milner, "A Calculus of Communicating Systems," Springer-Verlag, Berlin/New York, 1980.
11. R. Milner, "Communication and Concurrency," Prentice–Hall, Englewood Cliffs, NJ, 1989.
12. R. Milner, The polyadic π-calculus: a tutorial, *in* "Logic and Algebra of Specification," Springer-Verlag, Berlin/New York, 1992.
13. R. Milner, J. Parrow, and D. Walker, A calculus of mobile processes, I and II, *Inform. and Comput.* **100** (1992), 1–77.
14. A. Philippou and D. Walker, On transformations of concurrent object programs, *Theoret. Comput. Sci.* **195** (1998), 259–289.
15. B. Pierce and C. Sangiorgi, Typing and subtyping for mobile processes, *Math. Structures Comput. Sci.* **6** (1996), 409–454.
16. Y. Sagiv, Concurrent operations on B^*-trees with overtaking, *J. Comput. System Sci.* **33** (1986), 275–296.
17. D. Sangiorgi, The name discipline of uniform receptiveness, *in* "Proceedings of ICALP'97," pp. 303–313, Springer-Verlag, Berlin/New York, 1997.
18. C. Tofts, "Proof Methods and Pragmatics for Parallel Programming," Ph.D. thesis, University of Edinburgh, 1990.
19. F. Vaandrager, On the relationship between process algebra and input/output automata, *in* "Proceedings of LCS'91," pp. 387–398, Springer-Verlag, Berlin/New York, 1991.
20. H. Wedekind, On the selection of access paths in a data base system, *in* "Data Base Management," pp. 385–397, North-Holland, Amsterdam, 1974.