(4)

LABORATORY FOR COMPUTER SCIENCE

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

AD-A222 116

MIT/LCS/TR-474

A HIERARCHICAL PROOF OF AN ALGORITHM FOR DEADLOCK RECOVERY IN A SYSTEM USING REMOTE PROCEDURE CALLS

Gregory D. Troxel

January 1990

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY 2b. DECLASSIFICATION / DOWNGRADING SCHEDULE	3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for Public Release; distribution is limited.
4. PERFORMING ORGANIZATION REPORT NUMBER(S) MIT/LCS/TR 474	5. MONITORING ORGANIZATION REPORT NUMBER(S) N00014-85-K-0168
6a. NAME OF PERFORMING ORGANIZATION MIT Lab for Computer Science 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION Office of Naval Research/Dept. of Navy
6c. ADDRESS *(City, State, and ZIP Code)* 545 Technology Square Cmabridge, MA 02139	7b. ADDRESS *(City, State, and ZIP Code)* Information Systems Program Arlington, VA 22217
8a. NAME OF FUNDING / SPONSORING ORGANIZATION DARPA/DOD 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)* 1400 Wilson Blvd. Arlington, VA 22217	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)*
A Hierarchical Proof of an Algorithm for Deadlock Recovery in a System Using Remote Procedure Calls

12. PERSONAL AUTHOR(S)
Gregory D. Troxel

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
Technical	FROM ____ TO ____	January, 1990	127

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	I/O Automata, hierarchical proofs, proofs, remote procedure calls

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

An algorithm for detecting and recovering from deadlock in a system using remote procedure calls is presented, along with a proof of correctness. The proof uses the I/O Automata model of Lynch and Tuttle, described in [LT88] and [LT87]. First, correctness conditions for the problem are given in terms of I/O Automata. Next, a high-level graph-theoretic representation of the algorithm is shown to be correct. Then a lower-level formulation of the algorithm, taking into account its distributed nature, is shown to be equivalent to the higher-level representation, and thus correct.

In giving the correctness conditions, we introduce *client automata*, which model the behavior of the user's program, and allow almost all details of this user program to be suppressed at both specification and proof time.

To simplify the proof of the high-level version of the algorithm, safety properties are proved with a simplified version of the algorithm. Then, the algorithm is transformed to the full version, and it is argued that the safety properties hold for the transformed version.

20. DISTRIBUTION / AVAILABILITY OF ABSTRACT ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Publications	22b. TELEPHONE *(Include Area Code)* (617) 253-5894 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted. All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

*U.S. Government Printing Office: 1985-507-047

Unclassified

19a A new technique that can be used either for expanding the number of algorithms to which a proof applies or for simplifying the proof that a lower-level algorithm solves the same problem as a higher-level one is presented. This is effected by underspecifying the predicates used in the preconditions of the I/O automata. This captures the concept that whether the algorithm takes a certain action under an intermediate range of conditions does not impact its correctness. This lack of specification can be carried through to the low-level algorithm, presumably making the job of the implementor easier or allowing more efficient code, since then at times arbitrary choices may be made. It can also be used to make showing that the low-level algorithm solves the same problem as the high-level one easier.

The proof of the liveness properties of the high-level version of the algorithm makes use of a metric on states of the algorithm. Rather than the conventional technique of claiming that the set of values of the metric is well-founded, we show that every subset of such values occurring in a particular execution of the algorithm is well-founded. This enables us to allow the user program to request an arbitrary but finite number of remote procedure calls with arbitrary arguments.

We then present the low-level version of the algorithm, along with specifications for the communications network, etc. used by it. A proof is presented showing that the low-level version of the algorithm (together with the network, etc.) is equivalent (from the point of view of the user's program) to the high-level version.

A Hierarchical Proof of an Algorithm for Deadlock Recovery in a System using Remote Procedure Calls

by

Gregory D. Troxel
S.B., Massachusetts Institute of Technology
(1987)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in
Electrical Engineering and Computer Science
and
Electrical Engineer
at the
Massachusetts Institute of Technology
January 1990

Signature of Author ______________________________
Department of Electrical Engineering and Computer Science
January, 1990

Certified by ______________________________
Nancy A. Lynch
Thesis Supervisor

Certified by ______________________________
Richard E. Harper
Charles Stark Draper Laboratory

Accepted by ______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

A Hierarchical Proof of an Algorithm for Deadlock Recovery in a System using Remote Procedure Calls

by

Gregory D. Troxel

Submitted to the Department of Electrical Engineering and Computer Science on March 29, 1990 in partial fulfillment of the requirements for the degrees of Master of Science in Electrical Engineering and Computer Science and Electrical Engineer

Abstract

An algorithm for detecting and recovering from deadlock in a system using remote procedure calls is presented, along with a proof of correctness. The proof uses the I/O Automata model of Lynch and Tuttle, described in [LT88] and [LT87]. First, correctness conditions for the problem are given in terms of I/O Automata. Next, a high-level graph-theoretic representation of the algorithm is shown to be correct. Then a lower-level formulation of the algorithm, taking into account its distributed nature, is shown to be equivalent to the higher-level representation, and thus correct.

In giving the correctness conditions, we introduce *client automata*, which model the behavior of the user's program, and allow almost all details of this user program to be suppressed at both specification and proof time.

To simplify the proof of the high-level version of the algorithm, safety properties are proved with a simplified version of the algorithm. Then, the algorithm is transformed to the full version, and it is argued that the safety properties hold for the transformed version.

A new technique that can be used either for expanding the number of algorithms to which a proof applies or for simplifying the proof that a lower-level algorithm solves the same problem as a higher-level one is presented. This is effected by underspecifying the predicates used in the preconditions of the I/O automata. This captures the concept that whether the algorithm takes a certain action under an intermediate range of conditions does not impact its correctness. This lack of specification can be carried through to the low-level algorithm, presumably making the job of the implementor easier or allowing more efficient code, since then at times arbitrary choices may be made. It can also be used to make showing that the low-level algorithm solves the same problem as the high-level one easier.

The proof of the liveness properties of the high-level version of the algorithm makes use of a metric on states of the algorithm. Rather than the conventional technique of

[illegible]

claiming that the set of values of the metric is well-founded, we show that every subset of such values occurring in a particular execution of the algorithm is well-founded. This enables us to allow the user program to request an arbitrary but finite number of remote procedure calls with arbitrary arguments.

We then present the low-level version of the algorithm, along with specifications for the communications network, etc used by it. A proof is presented showing that the low-level version of the algorithm (together with the network, etc.) is equivalent (from the point of view of the user's program) to the high-level version.

Keywords: I/O Automata, hierarchical proofs, proofs, remote procedure calls

Thesis Supervisor: Nancy A. Lynch
Title: Professor of Computer Science

Acknowledgments

I would like to thank Nancy Lynch and Rick Harper, my thesis supervisors, for their help, suggestions, and encouragement. I would like to thank Jennifer Welch for careful reading and extensive comments on parts of an early draft. I would like to thank the members of the Theory of Distributed Systems group of the Laboratory for Computer Science, especially Alan Fekete, Ken Goldman, and Mark Tuttle for encouragement and advice. I would also like to thank my parents and friends for their support and encouragement during the writing of this thesis.

This work was done in part at the Charles Stark Draper Laboratory under an internal research and development grant, and in part at the MIT Laboratory for Computer Science with support from the National Science Foundation under contract number CCR-8611442 and the Office of Naval Research under contract number N00014-85-K-0168.

Contents

Chapter 1
Introduction

This thesis presents an algorithm for detecting and recovering from deadlock in a remote procedure call system and a proof that this algorithm is correct.

1.1 Motivation

The target environment that motivates this work is the Fault Tolerant Parallel Processor (FTPP) under development at the Charles Stark Draper Laboratory (CSDL) in Cambridge, Massachusetts. The FTPP is described in detail elsewhere ([Har85] and [Har87]).

1.1.1 FTPP

The FTPP consists of many (the prototype has 16) Motorola 68020 processors, each with its own local memory, connected together by a fault tolerant network specifically developed for the FTPP. Lower-level layers of software make the system appear to the programmer as a collection of reliable processors (generally one-third to one-quarter the number of physical processors) connected by a reliable network.

The FTPP is designed for applications that require highly reliable high-throughput computation. The designers believe that it achieves higher performance than other existing processors that are as reliable as the FTPP, and higher reliability than other existing processors that have similar performance levels. These issues are, however, far beyond the scope of this thesis; see [Har85] for more information.

One probable application of the FTPP is for autonomous vehicle control (an aircraft, spacecraft, or submarine). In this case the computer would be responsible for all aspects of vehicle management, including control, guidance, navigation and destination planning. This requires approximating solutions to the Traveling Salesman Problem so that the computer can determine the next destination. Prior work ([Kim86]) indicates that a parallel processing environment would be useful in quickly obtaining approximate solutions.

1.1.2 RPC Model

The programming model of asynchronous functional remote procedure calls (RPCs) has been selected for the FTPP. Nonfunctional remote procedure calls are discussed in [LG85]. This programming model is discussed in [Tro87], and is summarized below. By asynchronous we mean that the process making an remote procedure call may continue to compute and even make additional remote procedure calls without waiting for the first to complete; this is the mechanism by which parallelism is made available to the programmer. By functional we mean that remote procedure calls may not have side effects.

The purpose of the programming model is to provide a framework for expressing what we shall refer to as the "user's computation." It also serves as a basis for expressing correctness conditions for the system that is provided for running programs formulated in terms of this model. Throughout the thesis we assume that only one program is run at a time, and only consider a single run of it. It should be clear that multiple programs could be run by bundling them into a super-program to be run at once, or sequentially by having the main program call each subprogram in turn. Thus, we will no longer address this issue. Also, we will assume that the text of procedures is always accessible; this is easy, since it is read-only.

The user describes the "user's computation" by specifying several procedures. Each procedure may compute (possibly with side effects on variables local to that procedure). It may request that other procedures be evaluated with specified arguments. These other procedures are then its *children*; it is their *parent*. Computation continues, however, after the request is made, rather than blocking until the answer is ready. Thus, many requests can be made without regard to how many have completed. No guarantees are made about the order in which the answers to the requests become available. The procedure is informed when a child completes, and the results are then available. It may notify the system that it has completed (providing its "answer"). The answer generated by a procedure must depend only on the arguments specified with it.[1]

These rules allow a tree of procedures to be generated, with vertices representing instantiations of procedures and edges representing the parent-child relationships. The system may evaluate the procedures specified by the multiple requests of a given procedure in parallel.

The term *depth* is the depth of the tree described above.

The environment for which the programming model is intended consists of reliable processors connected by a reliable network. Each processor is capable of executing a certain number of processes. This limit is derived from stack space and task control block memory consumption and the amount of memory available; it may be configurable but will not change during a run of a program. No secondary storage is available for paging or swapping. We are only concerned with the number of processors and the number of processes per processor, and shall consider them constant. Each process can evaluate one procedure at a time. Thus, there is a limit on the number of procedures that can be evaluated at a time. This is a major restriction imposed by the programming model, but it does not seem avoidable given the lack of secondary storage. This limit, does not, however, rule out recursion, but does constrain the maximum depth. Initial estimates are that there will be 4–16 processors,

[1]It might also depend on a random number generator, but the important point is that it does not depend on the internal state of any other procedure.

and 16–256 processes per processor. Space for procedure text is not counted in this analysis, since we assume it is statically allocated on each processor.

Comparison with other models

This model is quite different from the model used for work in nested transactions ([LMWF88]). In nested transactions, it is expected that data objects will be accessed by many higher-level transactions, and will have state across those accesses. Our model explicitly forbids independently existing data objects by requiring that the result of a remote procedure call be a function of its arguments, and not of any other state.

1.1.3 Possibility of deadlock

Consider a simple scheduler for such a system. This scheduler assigns a remote procedure call to a process if a free process exists, and conveys requests to commit (with results) to the parent of the committing remote procedure call. Now, consider a large computation tree being evaluated using this scheduler. There can easily arise a situation where all processes have remote procedure calls assigned to them, but no remote procedure call can request to commit because all have children that have been requested but have not yet committed. Thus, every remote procedure call can be waiting for more processes to become free so that a child can be evaluated, but no processes can become free, and the system can deadlock. In order to make the remote procedure call system useful, we must add a deadlock detection and recovery algorithm. In this way it will be possible to guarantee that a computation will actually terminate.

1.1.4 Alternatives to deadlock recovery

Alternatively, various forms of preassigning work to various processors to avoid deadlock can be considered. There are several considerations which prevent this from being done completely. One is that the number of processors on which the computation is run is not known until run time, and may change *during* a computation (as processors fail, for example). Processor failure is not contradictory with being reliable, since the system may have to disable a processor in a controlled manner in order to ensure continued reliable operation. This could occur if one processor out of three that were being operated as a triply redundant processor failed, since the remaining processors would not be able to tolerate a fault.

Another is that it is possible that the computation will be a search problem in which the depth of parts of the search tree will depend strongly on the input data, which again cannot be predicted until run time. While these arguments preclude total scheduling ahead of time, they should not be construed to mean that any such attempts are without merit; reducing the frequency of deadlock can certainly be expected to improve performance of the system, since as will be seen later the deadlock recovery algorithm functions by throwing away work and later redoing it.

Another possible approach is to constrain the user to write programs that never deadlock. We feel that it is simpler for the user to present him with a simple model such as ours, and guarantee to execute the program correctly if the simple limit of maximum depth is observed. Of course, efficiency concerns will dictate that the user should attempt to avoid deadlock situations in the normal case. However, assuming that the user is also expected to provide a correctness proof of his program, the task is made much easier because he will only need to show that the depth of the remote procedure call tree induced by his program is bounded (by the parameter the given machine can guarantee, depending on the amount of resources available), rather then presenting a proof that deadlock will not occur. This modularity of proofs should reduce the total amount of work needed to implement and prove correct several programs.

1.2 New algorithm

A distributed scheduling algorithm for the RPC system that detects and recovers from deadlock is presented. The algorithm guarantees that (under reasonable behavior by the user's program) any computation with a depth less than the total number of processes (on all processors) will complete. The existence of such an algorithm makes the verification of programs built using the RPC system significantly easier since it need only be shown that the depth is bounded by the process limit, rather than that the computation never deadlocks due to lack of space. This is important especially if the size of the target system is not known at verification time.

1.3 Proof

A proof of correctness for the algorithm is presented. The model used in the proof (as opposed to the RPC programming model) is I/O automata, described in [LT88]. We assume the reader is familiar with this work; if not, the reader is strongly encouraged to obtain and read a copy before attempting to read this thesis in detail. More information can be found in [LT87]. Parts of the liveness proof were inspired by [WLL88] and [Wel88].

Separation of safety and liveness

A technique for breaking up proofs is presented. First, safety properties are proved with a simplified version of the algorithm. Since this algorithm is less complex, the safety proof is less complicated and irrelevant details only needed for showing liveness are suppressed.

Then, the algorithm is transformed to the full version, and it is argued that the transformation preserves safety properties. The main technique of the transformation is to add preconditions to the actions of the automaton (when specified as actions with preconditions and effects sections). Since this merely removes transitions from the transition function of the automaton, every execution of the new automaton is an execution of the old, and all safety properties still hold *a fortiori.*

Underspecified predicates

Underspecified predicates are presented. This is a mechanism for leaving out unnecessary details in the high-level version of the algorithm. These details can then be given in the low-level version, or left unspecified there, to be given in the actual implementation.

This is effected by underspecifying the predicates used in the preconditions of the I/O automata. In this proof, this technique is confined to those preconditions added for the liveness section. This captures the concept that in certain states, we do not need to be able to tell whether the algorithm takes certain actions in order to show correctness. The underspecification is effected by placing constraints on the underspecified predicates, and allowing any predicate meeting these constraints to be substituted in. The proof then only uses the information provided by these constraints, and thus is valid for any predicate satisfying the constraints.

This lack of specification can be carried through to the low-level algorithm, presumably making the job of the implementor easier or allowing more efficient code, since then at times arbitrary choices may be made. Possibly, choices that have not been fully specified can be fine-tuned for efficiency on an empirical basis, while maintaining correctness. Providing this freedom may encourage the use of proofs of algorithms in system-building. Additionally, it can be used to make the task of showing that the low-level algorithm solves the same problem as the high-level one easier, by allowing one more fully to specify the behavior of the low-level algorithm in ways that might seem unnatural or hard to express at the high level. These two uses are, in some sense, the same, since an implementation can be regarded as an even lower-level version of the algorithm presented as the "low-level" algorithm.

Summary equivalence classes

The safety property proof makes use of a mapping of schedules of the automata modeling the user's computation onto a small set of summary classes. The safety properties are then concerned only with the class to which each execution of a client automaton maps. This allows us to abstract away the details of the user program while specifying and proving the safety properties.

Chapter 2
Problem Statement

In this chapter we specify correctness conditions for the deadlock recovery algorithm.

We define *client automata* as a means of modeling the behavior of a user's program. We define a notion of well-formedness that these automata preserve, and specify an allowable set of fair behaviors.

We define a tree-based naming scheme and organization for a collection of client automata used to model the user's computation.

We then define a trivial scheduler which, when composed with a client automaton at every vertex of the tree that is the basis of the naming scheme, has the fair behaviors desired. Since this trivial scheduler has no resource constraints, all requests by client automata to evaluate remote procedure calls may easily be satisfied. We define correctness of a deadlock recovery automaton in terms of the fair behaviors of the composition of the trivial scheduler and the client automata and an additional constraint on resource usage (since the entire point of the deadlock recovery automaton is to perform computation when the amount of resources available is insufficient to guarantee that deadlock will not occur).

2.1 Static computation substrate

The basic structure of a computation involving nested functional remote procedure calls is that of a tree. The vertices are possible remote procedure call instantiations in the user's program, and the edges are the caller/callee relationships between these.

This tree is used for two purposes. One is naming the client automata used to model the user's computation. The other is describing part of the state of the deadlock recovery automaton.

2.1.1 Naming scheme

Assume that there is some alphabet used in naming, and, for any node, each child is associated with a unique element of this alphabet (children of different parents may use the same elements). Without loss of generality, we shall assume that the numerals 1 through 9 are the first elements of this alphabet.

There is a pseudo-node that exists only for the purpose of making the first request for computation to the system. The node that makes this first request is named *super-root.*

The children of *super-root* are named *root-1*, *root-2*, etc. As will be required later, in any execution the *super-root* will call exactly one such child. Thus, we shall refer

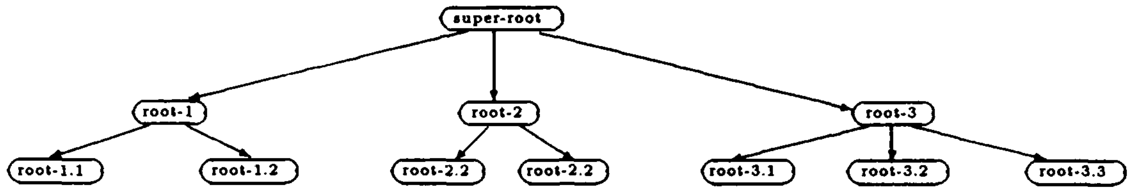

Figure 2.1: Tree structure

to this node simply as *root*.

The name of any other node is the name of its parent concatenated with the element of the alphabet associated with the child, separated by punctuation to make the names easier to read. Thus, the children of *root-1* are named *root-1.1*, *root-1.2*, etc. The children of *root-4.9* are *root-4.9.1*, *root-4.9.2*, etc. Figure 2.1 illustrates this convention.

We shall use the symbol T to refer to such a name.

Let *parent*(T) refer to the parent of T. The name of the parent of a node can be trivially obtained from the name of the node by removing the last component, with a special case when the parent is *super-root*.

Let *ancestors*(T) be the set containing T and all its ancestors in the computation tree. The node *super-root* is not considered to be an ancestor of any node, since it is meant to model the user requesting that the program be run, rather than part of the program itself.

Let *children*(T) be the set of nodes that are direct children of T.

2.1.2 Bound on depth of tree

We require the maximum depth of the tree to be less than TOTAL_RESOURCES, a parameter for the correctness conditions specifying the number of processes available to the deadlock recovery automaton. Thus, names T for which the number of components is greater than TOTAL_RESOURCES are not names of vertices of the tree.

Processes are a logical resource, capable of "executing" remote procedure calls.We assume that each process is identified with a processor (which has a network address, used in the distributed version). Several or many of our "processes" could be implemented by a single physical processor, but that issue is beyond the scope of this thesis.

There are a fixed number of processes, and each may be allocated to exactly one remote procedure call at a given time (or it may be idle). Any remote procedure call that has been created but not yet completed occupies a process. Thus, to evaluate a computation tree of depth h requires at least h processes. We therefore bound the depth of the computation tree, so that we only consider computations for which execution is possible.

2.2 Client automata

Client automata are used to model the behavior of the user's program. In modeling the user's program, the actual computations performed internally are of no concern to us; we are concerned with requests by client automata to create child remote procedure calls and to commit. Thus, we specify the external action signature for a client automaton, and define a notion of well-formedness for it. Then, we specify the fair behaviors of client automata.

2.2.1 External action signature

Client automata may request that children be evaluated or request to commit, and are informed when they are created or when the children they have requested have committed.

The external action signature of the client automaton for node T of the computation tree is given below. All actions of a given automaton are thus tagged with the name of the corresponding node in the computation tree (the argument T) to provide unique names for the actions of each automaton.

Input Actions:

create(T)

commit(T', results), T' a child of T

die(T)

Output Actions:

request-create-child(T'), T' a child of T

request-commit(T, results)

The following description of the meaning of the actions is merely intended to be helpful in understanding them, and should not be used when reasoning rigorously about the deadlock recovery algorithm. "Client" and "system" refer to client automata (or a particular one) and the deadlock recovery automaton, respectively. As specified above, the first three are input actions and the last two are output actions.

The create(T) action tells the client automaton named T that it should begin computation.

The **commit** action notifies the automaton that its child, specified by T', has committed with the return value *results*. The argument *results* is a description of the results of the computation.

The **die** action places the automaton back into a start state.

The **request-create-child** action, when taken by the automaton, is a request for computation (specified by T') to be performed. (In an implementation, the system might require the child procedure to be specified, and return a handle to the user by which it would refer to this child in the future.)

The **request-commit** action is taken by the client when it has completed computation, and the parameter *results* contains a description of the answer, to be passed to the parent via the **commit** action. The client automaton must be in a start state after this action.

2.2.2 Motivation

In the remote procedure call model, no provision is made for accessing global variables. This convention is automatically enforced by our model of client automata since the action signature of the client automata prevents any side effects or references to data of other automata.

The deadlock recovery automaton must be able to abort client automata when resources are needed elsewhere, and restart them later. In an implementation, this might be accomplished by recording the arguments when the RPC is initially started, effecting restart by discarding all state and restarting the procedure. To model this, a **die** input action is defined, and the client automaton is required to be reset to its initial state after this action.

We shall require that *super-root* only request one child; this seems to be a natural model.

2.2.3 Well-formedness for client automata

Here, we define well-formedness for sequences of external actions of client automata.

Definition:

A sequence of external actions of *super-root* is *well-formed* if it is the empty sequence, the single action **request-create-child**(T) or that action followed by **commit**(T, **results**).

Note that by the action signature of this client automaton, the child T requested must be one of the possible roots. The *super-root* may call exactly one "*root*" child, in keeping with our model that a user makes a single request for computation. Note that no infinite sequence of actions of *super-root* is well-formed.

The conditions for client automata other than *super-root* are given below.

Definition:

A block is a subsequence of external actions of a client automaton bounded by (but not including) either **die** actions or boundaries between **request-commit** and **create** actions.

Thus, the first block consists of all actions up to (but not including) the first **die** or all actions up to (and including) the first **request-commit** that is followed by **create**. For finite sequences, we may talk about the last block of a sequence. Note that a finite sequence ending in **request-commit** does not have an empty block at the end, but that a sequence ending in **die** does.

Definition:

Let *last-block*(β) be the last block of the finite sequence β, all actions after the last **die** or **request-commit/create** boundary. This will be the empty sequence if β is empty or ends in **die**.

Definition:

Two actions, **request-create-child**(T') and **commit**(T', **results**), are said to be *matching* if they are in the same block.

Definition:

A finite sequence of external actions β' of a (non *super-root*) client automaton is *well-formed* if for β, any block of any prefix of β', the following conditions hold:

- The first action of β is **create**(T), or β is empty.
- At most one **create** action appears in β.
- There is at most one matching **commit**(T', **results**) action for any **request-create-child**, and the **request-create-child** occurs before the commit action. No commit action for which there is no matching **request-create-child** action occurs.
- No **request-create-child**(T') action occurs more than once.
- If a **request-commit**(T, **results**) action occurs in β, then no actions occur after it, and every preceding **request-create-child**(T') action is followed by a matching **commit** action (which must precede the **request-commit**).

An infinite sequence is *well-formed* if every finite prefix of it is well-formed.

To summarize, a client automaton may only perform output actions after it has been created, may only **request-commit** if all children it has requested have returned, and may not request children after it has requested to commit.

Since the definition of well-formedness is hard to work with, we give an inductive characterization of finite well-formed sequences.

Definition:

Define the set W inductively as follows, assuming that T' is a child of T.

The empty sequence is in W. The sequence of actions $\beta' = \beta\pi$ is in W exactly if β is in W and the following hold:

- If π is **create**(T) then *last-block*(β) is empty, or ends in **request-commit**(T, **results**).
- if π is **request-create-child**(T') then **create**(T) occurs in *last-block*(β), **request-create-child**(T') does not appear in *last-block*(β), and no **request-commit** appears in *last-block*(β).
- if π is **commit**(T', **results**) then **request-create-child**(T') occurs in *last-block*(β), and no other **commit** action with the same identifier T' appears.
- if π is the action **request-commit**(T, **results**), then every **request-create** action in *last-block*(β) has a matching **commit** action, **create**(T) occurs in *last-block*(β), and **request-commit** does not.

Lemma 2.1 *W is exactly equal to the set of finite well-formed sequences of external actions of client automata.*

Proof:

We first show that every sequence in W is well-formed. We do this by induction on the length of sequences in W.

Basis:

Observe that the empty sequence, the basis of the inductive definition of W, is well-formed. It contains one block, which is empty, and thus this sequence satisfies the definition of well-formedness.

Inductive Step:

Suppose $\beta = \beta'\pi$, where π is a single action. Assume $\beta \in W$ and that β' is well-formed. We verify that this implies that β is well-formed by enumeration of possible values of π.

die(T): Every block of β is the same as a block of β', by the definition of blocks. Since β' is well-formed, β is well-formed.

create(T): Since β'**create**(T) $\in W$, *last-block*(β') is empty or ends in **request-commit**(T, **results**). Since the last block of β contains only **create**(T), and every other block is also in β', β is well-formed.

request-create-child(T'): Since $\beta \in W$, **create**(T) occurs in *last-block*(β') and **request-create-child**(T') and **request-commit**(T, **results**) do not. Examining the definition of well-formedness, we see that the first three requirements (pertaining to **create** and **commit**) are still satisfied. The next, uniqueness of **request-create-child** actions, is satisfied because we showed that **request-create-child**(T') did not exist. The last is satisfied because no **request-commit** action appears. Thus β is well-formed.

commit(T', **results**): Since β'**commit**(T', **results**) $\in W$, we know (by the definition of W) that **request-create-child**(T') occurs in *last-block*(β') and no other **commit** with the same identifier occurs. Since β' is well-formed, we know that **request-commit**(T, **results**) does not appear. Since the other conditions for well-formedness must be true of β if they are true of β', β is well-formed.

request-commit(T, results): Since $\beta \in W$, *last-block*(β') contains **create**(T), does not contain **request-commit**(T, **results**), and every **request-create-child** has a matching **commit**. Examining the definition of well-formedness, we see that the first four requirements (pertaining to **create**, **commit** and **request-create-child**) are still satisfied. The last is satisfied because no **request-commit** action exists in β' and every **request-create-child** has a matching **commit**. Thus, β is well-formed.

Thus, all sequences in W are well-formed.

Now we show that every finite well-formed sequence is in W.

Consider any finite well-formed sequence. By the definition of well-formedness, any prefix is also well-formed. By induction on the length of finite well-formed sequences, we show that every finite well-formed sequence is in W.

Basis:

Consider the smallest well-formed sequence, the empty sequence. This is explicitly included in W as the basis of the inductive definition.

Inductive step:

We must then show that if γ is in W and well-formed, if $\gamma\pi$ is well-formed, then $\gamma\pi$ is in W.

We consider each possible action π, and argue that either γ or $\gamma\pi$ is not well-formed or that $\gamma\pi$ is in W.

die(T): The definition of W, by failing to set conditions under which **die** can be added, includes in W any sequence in W extended by **die**(T). Thus, γ**die**(T) is in W.

create(T): If the last block of γ is empty, then γ**create**(T) is in W. If the last block of γ ends in **request-commit**(T, **results**), then γ**create**(T) is in W. If the last block of γ is non-empty and does not end in **request-commit**(T, **results**), then γ**create**(T) is not well-formed because the last block of it contains two **create** actions.

request-create-child(T'): If γ**request-create-child**(T') is well-formed, the last block of γ may not contain another **request-create-child**(T') action, or a **request-commit** action. Also, the first action of the last block must be **create**. Observing the conditions specified in the definition of W for adding **request-create-child**, we see that they are satisfied, and, therefore, γ**request-create-child** is in W.

commit(T', **results**): If γ**commit**(T', **results**) is well-formed, the last block of γ may not contain another **commit**(T', **results**) and must contain a matching **request-commit**. Since γ is in W and the conditions in the definition of W for adding **commit** are satisfied, γ**commit**(T', **results**) is in W.

request-commit(T, **results**): If γ**request-commit**(T, **results**) is well-formed, *last-block*(γ) must contain a matching **commit** action for every **request-create-child** action. Also, the last block must begin with **create**. Since no action may occur after a **request-commit** action, there is no **request-commit** action in γ. We see from the definition of W that γ**request-commit**(T, **results**) is in W.

Thus, all finite well-formed sequences of external actions of client automata are in W.

□

Recall that an infinite sequence is well-formed if every finite prefix of it is well-formed. Since W is the set of finite well-formed sequences, it is equivalent to state that an infinite sequence is well-formed if every finite prefix is in W. The inductive characterization just shown to be equivalent to the definition of well-formedness will generally be used in proofs.

Corollary 2.2 *The set of well-formed sequences of a client automaton T is non-empty, prefix-closed and limit-closed.*

Proof:

These properties obviously hold for ***super-root***, all well-formed sequences of which have length 0, 1 or 2.

For any other client automaton, observe that the empty sequence is well-formed, so the set of well-formed sequences is not empty.

Consider a well-formed sequence β, and γ, a prefix of β. If β is finite, $\beta \in W$, and therefore $\gamma \in W$ and is well-formed. If β is infinite, then every prefix of it (including γ) is well-formed, according to the definition of well-formedness. Thus, the set of well-formed sequences is prefix-closed.

Since an infinite sequence is well-formed exactly if every finite prefix is well-formed, the set of well-formed sequences is limit-closed.

□

2.2.4 Behaviors of client automata

In order for the deadlock recovery automaton to work correctly, we must preclude "unreasonable" actions on the part of client automata. In this section we place restrictions on behaviors of client automata.

Definition:

A ***client automaton*** C is an automaton with the external action signature given earlier such that:

- C preserves[1] well-formedness.
- In any well-formed behavior β of C there are a finite number of **request-create-child** actions in any block.
- Every fair behavior β of C satisfies the following implication:

  If β is well-formed and finite, then ***last-block***(β) either

  1. is empty, or

[1][LT88j], p. 13. C preserves well-formedness if $\beta\pi$ is well-formed whenever β is well-formed, π is an output action of C, and $\beta\pi$ is a finite behavior of C.

2. has a request-create for a child with no matching commit, or
3. contains a request-commit.

Restated informally, the second condition requires that a client automaton never requests an infinite number of children. The third condition means that a client automaton must continue to take steps unless it has not been created, has requested a child that has not committed, or has requested to commit. These conditions are intuitively satisfying since they mean that the deadlock recovery automaton will not be considered incorrect because the user's program requests an infinite number of children or unreasonably stops taking steps.

Lemma 2.3 *Every well-formed behavior of a client automaton that does not contain a* die *action is finite.*

Proof:
Since the behavior is well-formed and has only one block, it contains at most one create(T) and one request-create(T). By the definition of client automata, it has finitely many request-create-child(T') actions. Further, it contains at most one commit(T', results) action for each request-create. There are no die actions. Since there is not an infinite number of any action, the total number of actions is finite. □

2.3 Correctness conditions

Given that we have an automaton that implements the algorithm, referred to as the *deadlock recovery automaton*, we define correctness in terms of the composition of the deadlock recovery automaton and the client automata, referred to as A_h. The subscript h appears because this is the high-level version of the algorithm, as opposed to the distributed version presented in Chapter 6.

2.3.1 Composition

There is one client automaton at each node in the computation tree, including *super-root.* There is another automaton, the *deadlock recovery automaton*, that performs the deadlock detection and recovery function.

Definition:
A_h is the composition of the deadlock recovery automaton and all of the client automata.

In order to discuss the composition of these automata, we must show that they are *strongly compatible* ([LT88], p. 6). Without loss of generality, we can assume that any internal actions of the deadlock recovery automaton have names distinct from any action of any client. We must require that no output action of the deadlock recovery automaton is an output action of any client. In our case, it will be obvious from inspecting the automaton definition that this is true. Since no action of any client automaton is shared by any other client automaton, it is clear that there is no action shared by an infinite number of components.

2.3.2 Motivation

Intuitively, the deadlock recovery automaton must create all requested children, and must faithfully convey requests to commit back to the parents. This is hard to directly capture formally in a way that everyone will agree is reasonable. Thus, we define a trivial scheduler automaton. This scheduler has no constraints about resources and is extremely simple; we feel that everyone will be able to see that it "solves" the problem at hand.

We require that in an execution of A_h (the composition of the deadlock recovery automaton and client automaton, as defined above) the behavior at each client automaton after any die actions is one that could have occurred with the trivial scheduler automaton. Informally, the user's program should not be able to tell that it is interacting with a deadlock recovery automaton rather than a trivial scheduler.

2.3.3 Trivial scheduler

The state of the trivial scheduler consists of a variable *status* for each remote procedure call, and a set *save-results* of ordered pairs of remote procedure call names and their return values. The allowable values of *status* are *not-started*, *requested*, *running*, *committing*, and *committed*. These correspond to nodes in their initial state, nodes that have been requested by their parents, have been created, have requested to commit, and have been committed, respectively.

In the initial state each variable *status* has the value *not-started*, and the set *saved-results* is empty.

We place all output actions of this scheduler in the same class, and thus any finite behavior in which no locally-controlled action is enabled in the final state is fair. We later show that every fair behavior of the composition of the trivial scheduler and client automata is finite, and thus are not concerned with which infinite behaviors are fair.

Input Actions:

```
request-create-child($T$):
    Effects:
        status($T$) := requested

request-commit($T$, results):
    Effects:
        status($T$) := committing
        saved-results := saved-results $\cup$ ($T$, results)
```

Output Actions:

```
create(T):
  Preconditions:
    status(T) = requested
  Effects:
    status(T) := running

commit(T, results):
  Preconditions:
    status(T) = committing
    (T, results) ∈ saved-results
  Effects:
    status(T) := committed
```

We shall refer to the composition of the trivial scheduler and client automata as A_t.

Definition:

Let *triv-beh* be the set of fair behaviors of A_t.

Before proving properties of *triv-beh*, we informally defend the correctness of the trivial scheduler. We see that after a child is requested, the create action becomes enabled and remains so until it occurs. When a child requests to commit, the commit action becomes enabled and remains so until it occurs; the return value is associatively stored with the transaction identifier and thus faithfully passed to the parent. Since (as will be shown) there are no infinite behaviors in *triv-beh*, and since in any finite fair behavior no actions are enabled in the last state (by the definition of fair), the system satisfies every request made by a client.

Definition:

A behavior β of the composition of a scheduler and client automata is *well-formed* if $\forall T \beta|T$ is well-formed.

Lemma 2.4 *Every element of* triv-beh *is well-formed.*

Proof:

It suffices to show that every finite behavior is well-formed, since by definition an infinite behavior is well-formed if every finite prefix of it is well-formed.

Basis:

The empty behavior is well-formed.

Inductive Step:

Assume that $\beta \in$ *triv-beh* is well-formed, and that $\beta\pi \in$ *triv-beh*. Observe that $\beta\pi$ is well-formed by enumerating the possible values of π for arbitrary β and T.

request-create-child(T): Client automata preserve well-formedness, so this is obviously well-formed.

request-commit(T, results): As above, client automata preserve well-formedness.

create(T): According to the definition of W, the alternate characterization of well-formedness, **create(T)** may be added iff the current block of T is empty (or ends in **request-commit(T, results)**). The precondition for this action is that *status*(T) = *requested*, and we see that this can only be true if the **request-create-child(T)** action was the most recent action with identifier T. Since **request-create-child(T)** can only happen once (because client automata preserve well-formedness), **create(T)** can only occur once, and hence does not occur in β. Since **create(T)** does not occur in β and β is well-formed, $\beta|T$ is empty and $\beta\pi$ is well-formed.

commit(T, results): According to the definition of W, **commit(T, results)** may be added if **request-create-child(T)** appears and no **commit** with the same identifier appears. The precondition for this action is that *status*(T) = *committing*, and we see that this can only be true if the **request-commit(T, results)** action was the most recent action with identifier T. Since **request-commit** with a particular identifier can only occur once, by client automata well-formedness, no **commit** action can have occurred with this identifier. Since **request-commit(T, results)** occurs, **create(T)** occurs (by well-formedness). Since **create(T)** occurs, **request-create(T)** must occur, by inspection of the automaton code. Since **commit(T, results)** appears only once and is preceded by a matching **request-create(T)**, $\beta\pi$ is well-formed.

□

Lemma 2.5 *Every element of* triv-beh *is finite.*

Proof:

Every action of any element of ***triv-beh*** is an action of some client automaton. Since every element of ***triv-beh*** is well-formed (by the previous lemma), the number of actions at any client automaton is finite, by Lemma 2.3. Consider the set of all clients that take a non-zero number of actions. By well-formedness, the first action at each such client automaton T must be **create(T)**, and, therefore, **request-create(T)** must occur. Thus, any such client automaton is requested by its parent, and this set is a tree. Recall that the depth of the tree is at most TOTAL_RESOURCES, a constant. Hence, we have a tree with finite depth in which no vertex has an infinite number of children. By König's Lemma[2], the number of vertices in such a tree is finite.

Since the number of actions taken by any client is finite, and only finitely many clients take a non-zero number of steps, the total number of actions is finite. □

[2] [Eve79], p. 32.

2.3.4 Correctness in terms of the trivial scheduler

We give a specification of the deadlock recovery problem as a set of behaviors (*correct-beh*), and define deadlock recovery automaton correctness in terms of this set.

We note that a schedule of A_h may have **die** actions, while the trivial scheduler may not. We must, however, define *correct-beh* in such a way that the fair behaviors of A_h can be contained in *correct-beh.* To achieve this, we shall define a set of expanded schedules that include die actions, yet still correspond to *triv-beh.*

Definition:

Define *last-blocks(β)* for β a behavior of A_h to be the subsequence γ of β containing, for each T, exactly those actions in *last-block($\beta|T$).*

Definition:

Let *die-triv-beh* be the set of all behaviors β of A_h such that *last-blocks*(β) $\in$ *triv-beh.*

Note that sequences in *die-triv-beh* may have arbitrary actions occurring at a client T if those actions are followed by a **die**(T) action.

A schedule of A_h must not "use" more resources than are available; executing computations in a resource-poor or possibly resource-poor environment is the reason to construct a deadlock recovery automaton.

Definition:

Let the function *occupies-resources* of a sequence of external actions β of a client automaton be 1 if in *last-block*(β) a **create** but not a **request-commit** exists, and zero otherwise Let the function *max-resources* of γ, a sequence of external actions of A_h, be the maximum over all prefixes of γ of the number of client automata T such that *occupies-resources*($\gamma|T$) is 1, or

$$\max_{\gamma'\in\gamma}\sum_{T} \textit{occupies-resources}(\gamma'|T).$$

Definition:

Let *correct-beh* be the set of sequences, β, consisting of external actions of A_h such that the following are true.

1. β is finite
2. β is well-formed
3. *max-resources*(β) $\leq$ TOTAL_RESOURCES
4. $\beta \in$ *die-triv-beh*

Since there is only a finite amount of computation to be done, it is reasonable to require that it be done in a finite number of steps. By making this restriction explicitly, we avoid the problem of having to define *last-block* for an infinite schedule.

We require that every schedule be well-formed. Without this restriction, a deadlock recovery automaton that takes an action that violates well-formedness at a client automaton, but then later takes a die action for that client and thereafter does not violate well-formedness would be considered correct. However, we require that the RPC "operating system", which is the deadlock recovery automaton, never do this on the grounds that users should not have to deal with such unreasonable actions.

Definition:

A deadlock recovery automaton is *correct* if the fair behaviors of the composition of it and client automata are contained in *correct-beh.*

2.3.5 Sufficient conditions for correctness

Although the specification of correctness in terms of the trivial scheduler is easy to understand, it is less convenient to prove that a deadlock recovery automaton is correct using the definition directly than it is using an alternative characterization of correctness. We define an alternate set of schedules, and show that every every fair behavior of this set is in *correct-beh.* Thus, any deadlock recovery automaton for which the fair behaviors of A_h are contained in *alt-beh* is correct.

Definition:

Let *alt-sched* be the set of sequences β of external actions of A_h such that the following are true.

1. β is finite
2. β is well-formed
3. $\textit{max-resources}(\beta) \leq$ TOTAL_RESOURCES
4. Every **create**(T) action is preceded by a **request-create-child**(T) action and no **die**($\textit{parent}(T)$) action exists between the **request-create-child**(T) and the **create**(T).
5. Every **commit**(T, **results**) action is preceded by a **request-commit**(T, **results**) action with no intervening **die**(T) action.
6. For every **request-create-child**(T) later than any action **die**($\textit{parent}(T)$), there is a **create**(T) action following it, which itself is not followed by a **die**(T) action.
7. For every **request-commit**(T, **results**) later than any **die**($\textit{parent}(T)$), there is a (matching) **commit**(T, **results**) action following it.
8. For every T, $\beta|T$ is a fair behavior of client automaton T.

Definition:

Let *alt-beh* be the behaviors of *alt-sched.*

Lemma 2.6 alt-beh $\subseteq$ correct-beh

Proof:

Consider any element β of ***alt-beh***.

First, we note that it trivially satisfies the first three conditions for membership in *correct-beh* because they are also the first conditions for membership in *alt-beh*.

We must show that $\beta \in$ ***die-triv-beh***. Let γ be ***last-blocks(β)***. We claim that this behavior γ is a fair behavior of A_t, and thus $\beta \in$ ***die-triv-beh***. First we argue that it is a behavior of A_t, and then that it is a fair behavior.

Consider a particular client T. Since $\beta|T$ is well-formed, $\gamma|T$ is well-formed, since the definition of well-formedness requires all blocks to meet a set of conditions, and $\gamma|T$ is merely the last block of $\beta|T$. Thus, γ is well-formed for all clients.

To show that γ is a behavior of A_t, we must show that each action of γ is enabled when taken. We proceed by induction on the length of γ, using the fact that γ is well-formed and that the conditions on β required for membership in *alt-beh* are satisfied.

Basis:

The empty behavior is obviously a behavior of A_t.

Inductive Step:

Assume δ is a behavior of A_t, and $\delta\pi$ is a prefix of an element of *alt-beh*. We verify that $\delta\pi$ is a behavior of A_t by enumeration of the possible values of π:

request-create-child(T): This is an input action and thus always enabled.

request-commit(T): This is an input action and thus always enabled.

create(T): The conditions for membership in *alt-beh* require that there exists in β a request-create-child(T) action that is not followed by a die(*parent*(T)) action. Thus the request-create-child(T) action must be in γ (recall that request-create-child(T) is in the action signature of automaton T). Since the request-create-child(T) action occurs in γ, we know that the create(T) action had been enabled. Because γ is well-formed and thus create(T) may not appear twice, we know that create(T) has not occurred. Since create(T) remains enabled until it occurs, it must still be enabled.

commit(T, results): The conditions for membership in *alt-beh* require that there exists in β a request-commit(T, results) action that is not followed by a die(T) action. Thus, the request-commit(T, results) action must be in γ. Since the request-commit(T, results) action occurs in γ, we know that commit(T, results) had been enabled. Because γ is well-formed and thus commit(T, results) may not appear twice, we know that commit(T, results) has not occurred. Since commit(T, results) remains enabled until it occurs, it must still be enabled.

We must now show that γ is a fair behavior of A_t. We know that for all T, $\gamma|T$ is a fair behavior of T, because $\beta|T$ is a fair behavior of T (by the definition of *alt-sched*) and $\gamma|T =$ *last-block*($\beta|T$).

Construct from γ an execution α as follows. Take an execution α_T of each client automaton T such that *behavior*(α_T) $= \gamma|T$; there must be one, since otherwise $\gamma|T$ could not be a behavior of T. Modify γ by inserting each internal action of client T (from α_T)

immediately before the next external action of the client, and placing internal actions that are not followed by an external action at the end of the sequence. Call this new sequence of actions δ.

Now insert states between every two adjacent actions of δ, and at the beginning and end. For each client automaton, use the state from the corresponding position of α_T. The start state of the trivial scheduler automaton is given with the automaton description, and the state following each action can be straightforwardly obtained from the previous state by examining the automaton description. We call the resulting execution α; it is an execution of A_t (by the previous inductive proof, every locally-controlled action of the trivial scheduler is enabled when taken).

Since β is finite, there are a finite number of external actions of A_t in γ, although there might be an infinite number of internal actions of client automata. We note that internal actions of client automata cannot change the state of the trivial scheduler automaton, and thus if no locally-controlled action of the trivial scheduler is enabled in the state following the last external action of the trivial scheduler, no such action ever becomes enabled again and the execution is fair to the trivial scheduler.

We consider all the output actions (there are no internal actions) of the trivial scheduler, and show that each cannot be enabled in the state following the last such action.

create(T): If **request-create-child(T)** does not appear in γ, this action is not enabled. If it does, then by the conditions for membership in *alt-beh* there is a **create(T)** action in β not followed by a **die(T)** action, and thus the **create(T)** action is in γ.

commit(T, results): If **request-commit(T, results)** does not appear in γ, this action is not enabled. If it does, then by the conditions for membership in *alt-beh* there is a **commit(T, results)** in β which is not followed by a **die(T)**, and thus **commit(T, results)** is in γ, and thus not enabled.

Thus, α is a fair execution of the trivial scheduler. Since for all T, $\alpha|T$ is a fair execution of T, α is a fair execution of A_t. Since the behavior of α is γ, γ is a fair behavior of A_t, and is thus in *triv-beh*. Therefore, $\beta \in$ *die-triv-beh*. This is the last of the conditions for membership in *correct-beh*, and thus our arbitrary member of *alt-beh* is contained in *correct-beh*.

□

Lemma 2.7 *A deadlock recovery automaton is correct if its external action signature is strongly compatible with client automata and every fair behavior of the composition of it and client automata is contained in* alt-beh.

Proof:

Consider a deadlock recovery automaton for which every fair behavior of the composition of it and client automata is contained in *alt-beh*. Since *alt-beh* $\subseteq$ *correct-beh*, the fair behaviors of the above composition are contained in *correct-beh*. □

Chapter 3
Safety Version of the Deadlock Recovery Algorithm

In order to make the proof easier to understand, we present several different versions of the deadlock recovery automaton. The most important distinction is drawn between the high-level versions, in which there is one automaton operating on a single global state, and the distributed version, presented in Chapter 6, in which there is one automaton for every client automaton.

Additionally, two versions of the high-level version are presented. The first, or "safety" version, presented in this chapter, satisfies all of the required safety properties. However, it does not satisfy the liveness properties. The second, or "liveness" version, presented in Chapter 5, has more constraints about the behavior of the algorithm that seem to be necessary to prove the liveness properties. Although it does not possess the desired liveness properties, the safety version is simpler and thus it is easier to show that it possesses the desired safety properties. Later, we show that all safety properties that hold for the safety version (all safety properties — not just those we show) also hold for the liveness version.

We give an English description of the algorithm. We then describe the state of the deadlock recovery automaton, and then describe the automaton itself using preconditions/effects notation.

3.1 Algorithm description

The description of the algorithm given here is meant to be informative and explanatory. The explanation of what the algorithm does and the informal description of why it works is divided into two parts. The first part describes "normal operation", a subset of the actions of the deadlock recovery automaton that makes no attempts to detect, avoid, or recover from deadlock; this subset would only be able to run a given computation successfully if it happened to never run out of resources. The second part describes actions associated with deadlock detection and recovery. Successively more complex descriptions of the algorithm are given for this part — later versions will specify more precisely actions that are left indeterminate in the earlier ones.

3.1.1 Normal operation

When a process (represented by a node) requests the calling of another process (represented by a child of that node), the child is marked as pending creation. If there is space available, a node pending creation can be allocated space on some processor, thus "creating" the child.[1] When a process completes, the corresponding node is marked as having requested to commit. At any time the system may "commit" any node that is marked as having requested to commit. This act notifies the calling process of the completion of its child process and its return value.

This naïve scheduler, while able to complete the computation of a given problem if the number of processors is large enough, can lead to deadlock. Assume the user's computation requires each process to call b child processes, up to a depth of h. Now, assume that the creation of children proceeds in a balanced manner, so that all nodes at depth $h - 1$ are created before any are created at depth h. This will require at least b^{h-1} processes. Now, if this much space is not available (likely for moderate values of h, since this is exponential in h), nodes will not be able to be created, but merely to be requested. However, no process can complete (request to commit) since it has children that have been requested that cannot commit since they have not been created. Thus, the system is in deadlock.

3.1.2 Detection and recovery

Given that deadlock may happen, the algorithm must be able to detect and recover from deadlock. We do not require, however, that the algorithm detect deadlock before taking actions we think of as deadlock recovery.

When the system has detected the possibility of deadlock, it chooses a process and then aborts it. *Aborting* a node consists of killing that node and all of its descendants, and recording that the node was aborted. *Killing* a remote procedure call consists of discarding any in-progress computation of that node; we then say that that node has *died.* This action frees resources, and thus more processes may be created. At the time of abort, the node is marked so that it may be restarted at a later time when more resources are available. These steps are repeated if necessary.

An entire subtree rooted at the deadlock master is killed as part of deadlock recovery. The technique used to implement this is called "convergecast", and is discussed in [Seg83] and [Awe85]. The deadlock master is marked as ready to die, and this marking is propagated along the edges of the tree. When a node with the "ready to die" marking is able to die (having no remaining children is the main condition), it does so. When all the children of the parent node have died (or otherwise terminated), that node may die. In this manner all descendants of the deadlock master eventually die, and then the deadlock master may do so as well. This approach allows reasonable efficiency, a simple distributed implementation, and avoids having to synchronize actions across many processors in the distributed case.

If this strategy were applied naïvely, the system could alternately abort a node and restart it an infinite number of times, a clearly unacceptable result. We impose conditions on the choice of which process to abort and when processes may be aborted

[1]In the algorithm as presented we just count the amount of space assigned and do not worry about its location, as this algorithm assumes a load-balancer is available to it.

in order to prevent oscillations in the system and ensure progress. The main concept used by the algorithm to ensure progress is that of unabortable work, by which we mean work that cannot be aborted by the deadlock recovery automaton. As the computation proceeds and recovery actions take place, certain processes are designated "precious" (initially the root of the computation tree is the only node so designated). These processes never die, and thus any process that commits to a precious process has completed work that is not abortable.

The concept of unabortable work is the motivation behind many of the decisions in the design of the algorithm. Many details exist to preserve invariants necessary for showing that unabortable work is not undone. Others exist to create liveness properties to ensure that the amount of unabortable work that has been done will always increase.

Precious nodes are never aborted. Since a node's work is lost if any ancestor is aborted, we must also ensure that no ancestor of a precious node is ever aborted. Thus, we enforce the invariant that in every state the parent of every precious node is also precious, and thus that all ancestors of a precious node are precious. This is enforced by requiring that a node that is precious remain so until it commits, and that only children of precious processes become precious.

If too many nodes become precious, however, further deadlock can arise for the same reasons as before. The algorithm must be able to complete a computation for which the depth is equal to the maximum number of processes that may be active at once. Consider a system that can only run n nodes at once, running a computation with depth n. The invariant that all precious nodes lie on a simple path from the root helps in showing that this is achievable, since otherwise a situation could occur in which n nodes are precious and none of them can complete since they are all waiting for children. This invariant can be preserved by having the deadlock recovery automaton only designate a node precious if the parent of the node is precious and none of the children of that parent are precious.

The details of the algorithm are constructed so that it is possible to show that given any situation more unabortable work will be completed. Since the only way that work is undone is by aborting nodes, and progress is made unless the system is in deadlock, the algorithm concentrates on ensuring that aborting nodes leads to the achievement of unabortable work. That this always occurs is the subject of Chapter 5, and the basic motivations are given here since they seem helpful in understanding the algorithm.

The first time deadlock is detected, a (non-precious) node is selected for aborting. This node is called the deadlock master. The next time, the parent of this node is aborted, and each time deadlock is detected the next ancestor is aborted. This continues until a precious ancestor is reached. Since a precious node cannot be aborted, a new node is chosen. Again (assuming deadlock continues to occur) the ancestors of this node are killed. Eventually, all the children of the precious node furthest from *root* are aborted. Since this node has aborted children and no "live" (children that have been requested but neither aborted nor committed) children, an aborted child is restarted. It is also made precious, thus increasing the number of precious processes.

After the number of precious processes increases to the point where a leaf process (one that has no children) is precious, this leaf child will commit (since it can request no children, it must commit), and unabortable work will have been achieved since its parent must also be precious.

3.2 The state of the deadlock recovery automaton

Because of the restricted way in which the liveness version of the high-level algorithm differs from the safety version, the set of possible states is identical. The state of the automaton consists of a static (with respect to both size and configuration) tree, along with information attached to each node and to the tree as a whole.

3.2.1 Description of the state

The tree corresponds exactly with the tree organization for the client automata discussed in Section 2.1 and uses the same naming scheme. We shall use the words vertex and node interchangeably, tending to use the former when we wish to emphasize properties of trees rather than remote procedure calls.

The state consists of variables *status* and *death-status* associated with each vertex, and the variables *master*, *precious* and *saved-results*, which are not associated with any vertex. Let *status*(T) be the value of the variable *status* associated with vertex T, and *death-status*(T) the value of *death-status*.

We describe each component of the state, and give its value in the initial state of the deadlock recovery automaton.

The variable *status*

Each node has a variable *status*, the possible values of which are enumerated below. A description of the meaning of *status*(T) having this value is also given; these are meant to help in understanding the formal description of the algorithm in terms of I/O automata and are not part of the specification of the algorithm.

not-started: This node is inactive; it either has not been requested, or its parent has died or committed.

requested: The parent of this node has requested that it be created. No resources have yet been allocated for nodes having this value of *status*.

running: The node is executing, and occupies resources.

committing: The node has requested to commit, and no longer occupies resources.

committed: The node has been committed.

dead: The node has died, and occupies no resources. This differs from *not-started* in that it must be restarted eventually.

In the initial state, the distinguished node *super-root* has *status* = *running*, while for all others it is *not-started.*

The variable *death-status*

This variable contains additional state relating to the die convergecast procedure; possible values are described below.

nil: In the initial state and during normal operation, *death-status* = *nil.* After a node has been killed, *death-status* also has this value.

dying: The node has been notified that it should die.

The variable *master*

Its value ranges over the set of nodes in the tree and *nil.* It contains the identity of the current deadlock master or *nil* if there is no deadlock master. In the initial state, *master* = *nil.*

The variable *precious*

precious is a set of nodes. A node's membership in this set means that it is precious. In the initial state, *precious* = $\emptyset$.

The variable *saved-results*

saved-results is a set of ordered pairs used to store the return values from remote procedure calls as they are committing. Each pair is of the form (T, *results*). In the initial state, *saved-results* = $\emptyset$

3.2.2 Notation for the tree state

The following are definitions, functions and derived variables of the state, introduced to make the algorithm code easier to follow and reason about.

Derived variables

Let *using-resources* be the set of all nodes for which

$$status(T) = running.$$

Let *alive-nodes* be the set of all nodes for which

$$status(T) \in \{requested, running, committing\}.$$

Let *alive-children(T)* be

$$alive\text{-}nodes \cap children(T).$$

Let *alive-children-count(T)* be

$$|\{alive\text{-}children(T)\}|.$$

Let *alive-child*(T) be the sole element of *alive-children*(T) in states in which *alive-children-count*$(T) = 1$, and be otherwise undefined.

Let *dead-children(T)* be

$$dead\text{-}nodes \cap children(T).$$

3.3 The deadlock recovery automaton

The following is the automaton definition for the safety version of the deadlock recovery automaton.

Input Actions:

```
request-create-child($T$), $T$ not a root node:
  Effects:
    status($T$) := requested
request-create-child($T$), $T$ a root node:
  Effects:
    status($T$) := requested
    precious := precious $\cup$ $T$

request-commit($T$, results):
  Effects:
    status($T$) := committing
    saved-results := saved-results $\cup$ ($T$, results)
    death-status($T$) := nil
    $\forall T' \in$ children($T$), status($T'$) := not-started
```

Output Actions:

```
create($T$):
  Preconditions:
    status($T$) = requested
    $\|$using-resources$\|$ < TOTAL_RESOURCES
  Effects:
    status($T$) := running

commit($T'$, results):
  Preconditions:
    status($T'$) = committing
    ($T'$, results) $\in$ saved-results
  Effects:
    status($T'$) := committed
    saved-results := saved-results $-$ ($T'$, results)
    precious := precious $-$ $\{T'\}$
    if master = $T'$
      if parent($T'$) $\in$ precious
        master := nil
      else
        master := parent($T'$)

die($T$):
  Preconditions:
    all children of $T$ have status $\in$ { not-started, requested, committed, dead }
    death-status($T$) := dying
  Effects:
    status($T$) := dead
    death-status($T$) := nil
    $\forall T', T' \in children(T)$, status($T'$) := not-started
```

Internal Actions:

```
restart($T$):
  Preconditions:
    $T'$ = parent($T$)
    $T$ $\in$ dead-children($T'$)
    status($T'$) = running
    death-status($T'$) = nil
  Effects:
    status($T$) := requested

restart-precious($T$):
  Preconditions:
    $T' = parent(T)$
    $T \in$ dead-children($T'$)
    status($T'$) = running
    death-status($T'$) = $nil$
    $T' \in$ precious $\wedge$ children($T'$) $\cap$ precious = $\emptyset$
  Effects:
    status($T$) := requested
    precious := precious $\cup$ $\{T\}$

become-master($T$):
  Preconditions:
    master = $nil$
    $T \notin$ precious
    status = running
  Effects:
    master := $T$

new-master($T$):
  Preconditions:
    master $\in$ children($T$)
    status(master) = dead
  Effects:
    if $T \in$ precious then
      master := $nil$
    else
      master := $T$

begin-die($T$):
  Preconditions:
    status = $running$
    death-status($T$) = $nil$
    master = $T$
  Effects:
    death-status($T$) = dying
```

```
die-down($T$):
  Preconditions:
    status($T$) = running
    death-status($parent(T)$) = dying
    death-status($T$) = $nil$
  Effects:
    death-status($T$) := dying

precious($T$):
  Preconditions:
    $T' \in$ precious
    $T \in$ alive-children($T'$)
    $T \neq$ master
    children($T'$) $\cap$ precious = $\emptyset$
  Effects:
    precious := precious $\cup \{T\}$
```

Chapter 4
Safety Properties

This chapter presents a proof that A_h possesses certain safety properties. It is argued that every behavior of A_h satisfies the first five, or "safety," conditions necessary for membership in *alt-beh*, described in Section 2.3.5 on page 30. More safety properties than are needed to show these properties are proved, however, because they are used in the proof of liveness properties.

4.1 Summary classes for client automata

By the definition of client automata, we know that any client automaton preserves well-formedness. This requirement often precludes a client automaton from taking a particular — or any — output action, and the algorithm relies on this. Further, we find that there is a correlation between the state of the deadlock recovery automaton and the set of locally-controlled actions a client automaton may take. Thus, we characterize behaviors of client automata so that we may reason about them more easily.

We define *summary classes*, and place each well-formed finite behavior of a client automaton into one of these classes. There is no need to consider behaviors that are not well-formed because we have already required that client automata preserve well-formedness and will argue that the deadlock recovery automaton preserves well-formedness (for client automata). We show that a behavior of a client automaton in a certain summary class may only be extended in particular ways. Then, we use summary classes in the expression of safety properties, abstracting away the details of client automata.

We define the summary classes by describing, for each class, the behaviors that belong to it. When we say that a client automaton is *in* a summary class at a particular time, we mean that the behavior of the client automaton is in the summary class. For example, a client automaton that has taken a **create** action and then a **request-create-child** action would be in the *child-outstanding* class.

Consider a client automaton A_c with finite behavior β'. We define below the summary class of the client automaton as a function of *last-block*(β'), which we will denote by β.

idle: β is empty or β contains **request-commit**.

started: β is non-empty, does not contain a **request-commit** action, and every **request-create-child** has a matching **commit** action.

	input actions			output actions	
class	**create**	**die**	**commit**	**req-cr-ch**	**request-commit**
idle	started	idle	dra-error	client-error	client-error
started	dra-error	idle	dra-error	ch-out	idle
ch-out	dra-error	idle	ch-out/started	client-error	client-error

Table 4.1: Transition table for summary classes.

child-outstanding: β is non-empty, does not contain a request-commit, and there is a request-create-child without a matching commit action.

It is easy to see that every finite well-formed behavior is in one of these classes. The empty behavior is in *idle*. Any behavior containing request-commit is also in *idle*. All other behaviors are obviously non-empty and do not contain request-commit and fall into *child-outstanding* if they contain an unfulfilled child request and *started* otherwise.

We give an informal characterization of these classes as an aid to understanding the safety proof.

idle: A client in this class either has not been created, has been created and requested to commit, or has been killed.

started: A client in this class has been created, and every child that it has requested has committed.

child-outstanding: A client in this class has been created and has requested at least one child that has not committed.

Table 4.1 is included to help the reader understand and to motivate summary classes. Given a behavior for which the summary class is given in the left column, the table shows which summary classes are possible after an action. The notations client-error and dra-error mean that the action would violate well-formedness, and that the client and the deadlock recovery automaton, respectively, are at fault. Note that this table is derived from the definitions and is *not* an equivalent definition because of the uncertainty in the summary class after a commit action.

4.1.1 Well-formedness constraints in terms of summary classes

We wish to rephrase some of the well-formedness constraints in terms of summary classes, so that we may use the fact that an automaton is in a particular summary class to rule out its taking certain actions. We also wish to be able to tell whether a client *occupies resources* by examining its summary class; this will make it easier to show that the system never occupies more resources than is allowed.

Lemma 4.1 *The following statements are true of every well-formed behavior of a client automaton:*

- **request-commit** only occurs when the finite behavior before it is in the *started* class.
- No locally-controlled action of a client automaton occurs when the finite behavior before it is in the *idle* class.

Proof:

By the definition of W (page 21) and the fact the W is the set of finite well-formed sequences of external actions of client automata (Lemma 2.1), **request-commit**(T, **results**) can only occur if every preceding **request-create** has a matching **commit** action, **create**(T) occurs and **request-commit** does not. Thus the conditions for the execution to belong to the *started* class are satisfied.

Assume a client is in the *idle* summary class, and has behavior β. Thus, *last-block*(β) is either empty or contains a **request-commit**. Examining the definition of W, we see that neither β**request-create-child** nor β**request-commit** is well-formed.

□

Lemma 4.2 *A client automaton* occupies resources *if and only if it is in either the* started *or* child-outstanding *summary class.*

Proof:

Examine the definitions of the summary classes and observe that exactly in the *started* and *child-outstanding* classes are the requirements for *occupying resources* met. In particular, note that the conditions for the *idle* class require either that the current block is empty (and therefore does not contain **create**) or contains **request-commit**. □

4.2 List of safety properties

Definition:

Let *restricted-last-blocks*(β) be similar to *last-blocks*(β) except that if the last block of any client contains a **request-commit** action no actions from that client appear in *restricted-last-blocks.*

This definition is made because some properties are more naturally expressed in terms of *restricted-last-blocks*, and some in terms of *last-blocks.*

Lemma 4.3 *The following statements are true of any execution* $\alpha = \ldots s_{n-1}\pi_{n-1}s_n$ *of* A_h.

1. *behavior(α)* is well-formed.
2. In state s_n of α, the following are true:

(a) $status(T) \neq not\text{-}started \iff$ a **request-create-child**(T) action exists in $restricted\text{-}last\text{-}blocks(\alpha)$.

(b) $status(T) = committing \wedge (T, results) \in saved\text{-}results \implies$ a **request-commit**(T, **results**) action exists in $last\text{-}blocks(\alpha)$.

(c) $status(T) = committing \iff$

$\exists\, results$ such that $(T, results) \in saved\text{-}results$

(d) $status(T) = committed \iff$ **commit**(T, **results**) exists in $restricted\text{-}last\text{-}blocks(\alpha)$.

3. In state s_n of α, the following are true for all T:

(a) $status(T) \neq not\text{-}started \implies status(parent(T)) = running$

(b) $T = master \implies status(T) \in \{running, committing, dead\}$

(c) $T = master \implies T \notin precious$

(d) $T \in precious \implies status(T) \in \{requested, running, committing\}$

(e) $T \in precious \implies ancestors(T) \subseteq precious$

(f) $|children(T) \cap precious| \leq 1$

4. In state s_n of α, the following are true for all T:

(a) $death\text{-}status(T) = dying \implies status(T) = running$

(b) $death\text{-}status(T) = dying \implies$

$death\text{-}status(parent(T)) = dying \oplus master = T$

(c) $status(T) = dying \implies master \neq nil \wedge death\text{-}status(master) = dying$

5. Let S_T represent the summary class of client automaton T in s_n. In state s_n of α, the following are true for all T:

(a) $S_T \in \{started, child\text{-}outstanding\} \iff status(T) = running$

(b) $S_T = started \implies$

$\forall c \in children(T) : status(c) \in \{not\text{-}started, committed\}$

(c) $S_T = child\text{-}outstanding \implies$

$\exists c \in children(T) : status(c) \notin \{not\text{-}started, committed\}$

Proof:

The proof of these properties is by induction on the length of executions.

Basis:

Consider the shortest execution possible, the one consisting of one state and no actions. In the initial state, it is easy to see that the conditions are satisfied.

Since the sequence consisting of no actions of a client automaton is well-formed, the first condition is satisfied.

All nodes have status *not-started*, except *super-root*, which has *status running*. Thus, the first group of properties hold. In the initial state, *master* = *nil* and *precious* = *nil*, and the properties of the next group are satisfied. No node has *death-status dying*, so the properties of the next group are satisfied.

All client automata are in summary class *idle*, since their schedules are empty. The corresponding nodes have status *not-started*. Thus, the properties in the last section hold.

Inductive step:

Given an execution $\alpha = \ldots \pi_{n-2} s_{n-2} \pi_{n-1} s_{n-1}$ that satisfies the conditions, we show that $\alpha' = \ldots \pi_{n-2} s_{n-2} \pi_{n-1} s_{n-1} \pi_n s_n$ satisfies the conditions by enumerating possible values of π_n. In general we will argue that each condition holds, using the same organization as the statement of the conditions. Conditions that are obviously unchanged by the action will not be discussed.

request-create-child(T), T not a root node:

1. Since clients preserve well-formedness, α' is well-formed.

2. Since in the previous state no **request-create-child**(T) action existed in *restricted-last-blocks*(α), *status*(T) must have been *not-started*.

   (a) After this action, *status*(T) is *requested*, and a **request-create-child**(T) action occurs, and thus the condition is true.

   (b) *status*(T) = *requested*, so this condition holds.

   (c) Since *status*(T) in s_{n-1} was *requested*, it was not *committing*, and thus the left hand side was false in both the previous and current states, and thus the condition is satisfied in the current state.

   (d) Since *status*(T) in s_{n-1} was *requested*, it was not *committed*, and thus the left hand side was false in both the previous and current states, and thus the condition is satisfied in the current state.

3. (a) In s_n *status*(T) = *requested*, and we must verify that *status*(*parent*(T)) = *running*. By Lemma 4.1, no locally controlled action can occur when a client automaton is in the *idle* class, and by the first point of the fourth statement, *status*(*parent*(T)) = *running*.

   (b) Observing that *status*(T) changed from *not-started* to *requested*, it is clear that the remaining conditions in this group hold.

4. It is clear that these conditions are still true given that they were true in the previous state.

5. The summary class of T remains *idle*, and its *status* does not become running, so the conditions hold for T. The summary class of *parent*(T) changes from *started* to *child-outstanding*, and in s_n *parent*(T) has a child with *status requested*, so the conditions hold for *parent*(T).

request-create-child(T), T a root node:

This action is identical to the previous one, except that T is added to *precious*. Since only four points (third through sixth) of the third statement relate to preciousness, we verify that they hold.

1. Since *status*(T) was *not-started* in the previous state, *master* $\neq T$ in the previous and hence current states, the condition holds.
2. *status*(T) is *requested* after the action, so this condition holds.
3. *ancestors*(T) is empty, and thus is trivially a subset of *precious*.
4. By client well-formedness, the schedule of *super-root* may only contain one request-create-child action, and thus all other children of *super-root* have *status* = *not-started* and are thus not precious, so the last point is satisfied.

request-commit(T, results):

1. Since client automaton preserve well-formedness, α' is well-formed.
2. Before this action, the summary class of client automaton T was *started*, and thus *status*(T) was *running*.
   (a) *status*(T) changed from *running* to *committing*, so the condition still holds for T. For any child T' of T, no request-create-child(T') exists in *restricted-last-blocks*(α) because *request-commit(T, results)* exists in α. *status*(T') for any such T' is *not-started*, by the postconditions of request-commit, so the condition holds.
   (b) *status*(T) is now *committing*, and request-commit(T, results) clearly exists in *last-blocks*(α).
   (c) This condition holds, by the postconditions of request-commit.
   (d) This condition holds, since *status*(T) = *committing* and was *running* in the previous state.
3. (a) Since *status*(T) changed from *running* to *committing*, this condition still holds for T. By the postconditions of request-commit(T, results), every child of T has *status not-started*, so the condition holds for every child of T.
   (b) *status*(T) = *committing*, so the condition still holds.
   (c) Since neither the value of *master* nor *precious* changed, this condition still holds.
   (d) Since the request-commit(T, results) action occurred, T must have been in the *started* summary class, and therefore every child of T had *status not-started* or *committed.* Thus, no child was a member of *precious*, and no child is in *precious* in s_n. Thus, this condition holds.
   (e) The next two conditions hold because they held in the previous state and the value of *precious* is unchanged.

4. Observe that *death-status*(T) is *nil*, by the postconditions of the action, and that *status* for each child T' of T was *not-started* or *committed* in the previous state since T was in summary class *started.* Thus, *death-status*(T') was *nil* in the previous state.

   (a) *death-status*(T) is *nil*, so the condition holds for T. *death-status* for each child of T was and thus still is *nil.* Thus, the condition holds.

   (b) This obviously holds for T, and holds for each child of T, since each child had *status not-started* or *committed.*

   (c) We must show that if *master* $= T$ no node has *death-status dying.* Assume *master* $= T$ and some other node has *death-status dying.* Because this condition held in the previous state, *death-status*(T) $=$ *dying.* We have already shown that no child of T has *death-status dying*, and no descendant of T has *death-status dying*, since then it would have *status running.* Consider an ancestor T' of T with *status*(T') $=$ *dying.* Since *master* $\neq T'$ and is not an ancestor of T', each ancestor of T' must have *death-status dying*, by induction on this condition. However, this condition cannot hold for *super-root* and thus no such node exists. But this contradicts our assumption, so the condition holds.

5. The summary class of T is now *idle*, and its *status* is no longer *running*, so the summary class conditions hold for T. They are unchanged for other nodes.

create(T):

1. Since *status*(T) was *requested* in the previous state, the summary class was *idle.* Thus, the last block of T either is empty or contains a **request-commit**(T, **results**), in which case that action is the last action of the block. Thus, the resulting execution is well-formed, since **create**(T) may be added if the current block is empty, and will start a block if it ends in **request-commit**.

2. (a) The first condition still holds, since *status*(T) changed from *requested* to *running.*

   (b) *status*(T) $\neq$ *committing*, so this condition holds.

   (c) This condition held in the previous state, and holds now, since *status*(T) changed from *requested* to *running.*

   (d) This condition held in the previous state, and holds now, since *status*(T) changed from *requested* to *running.*

3. (a) In the previous state, *status*(T) was *requested*, so *status*(*parent*(T)) was and is *running*, so this condition holds.

   (b) *status*(T) $=$ *running*, so this condition holds.

   The rest of the conditions can be easily observed to hold in s_n, given that they held in s_{n-1}.

4. (a) *status*(T) $=$ *running*, by the postconditions of the action, so this statement holds for T. It holds for all other nodes because it held in the previous state.

(b) Neither of the next two conditions are affected by the action, and thus still hold.

5. The summary class of T changes from *idle* to *started* as **create**(T) occurs, and *status*(T) becomes *running*. Since *status*(T) was not *running* in the previous state, all children of T had and have *status not-started*. Thus, the summary class conditions hold.

commit(T, **results**):

1. To show that α' is well-formed, we must show that **request-create-child**(T) occurs in *last-blocks*(α) and that **commit**(T, **results**) does not. The preconditions of **commit**(T, **results**) require that *status*(T) = *committing*, and therefore a **request-create-child**(T) action exists in *last-blocks*(α). By the fourth point of the second statement, no **commit**(T, **results**) action occurs in *restricted-last-blocks*(α), because *status*(T) $\neq$ *committed*. Because *status*(T) = *committing* in s_{n-1}, *status*(*parent*(T)) = *running*, and thus its summary class is either *started* or *child-outstanding*, and thus no **request-commit**(*parent*(T), **results**) exists in *last-blocks*(α). Thus, every action occurring at T in *restricted-last-blocks*(α) appears in *last-blocks*(α), and no **commit**(T, **results**). Thus, α' is well-formed.

2. (a) The left half of this condition is false both before and after the action, so it holds in s_n.

   (b) The antecedent is false, so the condition holds.

   (c) By the postconditions of **commit**(T, **results**), both sides of the equivalence are false in s_n.

   (d) By the postconditions of the action, *status*(T) = *committed*, and thus this condition holds.

3. (a) Since *status*(T) was *committing* in the previous state, *status*(*parent*(T)) was and is *running*, and the condition holds.

   (b) By the postconditions of the action, *master* is set to the parent of T if it was T. Since *status*(T) is not *not-started*, *status*(*parent*(T)) is *running*, and the condition is satisfied.

   (c) $T \notin$ *precious*, so the next three conditions hold.

   (f) Since *precious* now has either one fewer member or is the same, this statement still holds.

4. These conditions hold because they held in the previous state and do not depend on any variables changed by **commit**.

5. (a) *status*(*parent*(T)) was *running* before the action, and thus still is. The summary class of *parent*(T) is now either *started* or *child-outstanding*, and thus the first statement is satisfied.

   (b) If the summary class of *parent*(T) is *started*, then every **request-create-child**(T') has a matching **commit**, and no **request-commit**(*parent*(T), **results**) appears in *last-blocks*(α). By the last condition of the second statement, *status*(T') = *committed* if commit(T', results) appears in *last-blocks*(α). If no request-create-child(T') appears in *last-blocks*(α), then *status*(T') = *not-started*. Thus, every child of *parent*(T) has *status* either *not-started* or *committed*.

(c) If the summary class of *parent*(T) is *child-outstanding*, then there exists T', T' a child of T, such that a request-create-child(T') action with no matching commit appears in *restricted-last-blocks*(α). Thus, *status*(T') may not be *not-started*, and may not be *committed*.

die(T):

1. After die(T), the last block of T is empty, and thus well-formed. The last block of every other client automaton is unchanged, and is thus well-formed since it was well-formed before this action.

2. (a) For T' a child of T, no request-create-child(T') exists in *restricted-last-blocks*(α), since the last block of T is empty. By the postconditions of the action, *status*(T') = *not-started*, so the condition holds.

   (b) *status*(T) = *dead*, and *status*(T') = *not-started*, T' a child of T, so the condition holds.

   (c) By the preconditions of die(T), no child has *status committing*, so both sides of the equivalence are false before the action for each child of T. Since *status* of each child is *not-started* after, the condition holds for all children. Since *death-status*(T) = *dying* before the action, *status*(T) = *running*, and both sides of the equivalence are false before the action. Thus, they are false after the action, since then *status*(T) = *dead*.

   (d) *status*(T) changed from *running* to *dead*, so the statement still holds for T after the action. For any T', T' a child of T, *status*(T') = *not-started* and no commit(T', results) action occurs in *last-blocks*(α) because the last block of T is empty, and hence does not occur in *restricted-last-blocks*(α).

3. (a) Before the action, *status*(T) = *running*, and thus *status*(*parent*(T)) = *running*. After the action, *status*(*parent*(T)) is still *running*, so the condition holds for T. After the action, no child of T has *status running*, and the condition holds for each child.

   (b) By induction on the second condition of the fourth statement and because *death-status*(T) was dying, *master* is either T or an ancestor of T. *status*(T) = *dead*, and *status* for any ancestor of T is *running*, so the condition holds.

   (c) This condition holds because it held in the previous state because no node became *master* or was removed from *precious*.

   (d) Since *death-status*(T) = *dying*, *master* is an ancestor of T. Since *master* $\notin$ *precious*, no descendant of *master* is in *precious*. Thus, $T \notin$ *precious*.

   (e) The value of *precious* is unchanged, so the next two statements still hold.

4. (a) By the postconditions of the action, *death-status*(T) = *nil*, and the condition holds for T. For any child of T, *status* was not *running* and hence *death-status* was not *dying*. Thus, *death-status* is not *dying*, and the condition holds.
   (b) *death-status* is not *dying* for any chil !, as shown in the previous section. Thus, this condition still holds.
   (c) Observe that *master* $\neq$ *nil* in the previous state and hence the current state because *death-status*(T) = *dying*.

   If *master* $= T$, no node but T has *death-status dying*, since no child of T has *death-status dying*, and if there existed any ancestoı witl *death-status dying*, *master* would have to be an ancestor of this ancestor.

   If *master* $\neq T$, *death-status*(*master*) = *dying* in the previous state and thus also in s_n.

   Thus, the condition holds.

5. (a) After this action, the summary class of T is *idle*, and *status*(T) = *dead*, so the condition is satisfied. For every child T' of T, *status*(T') was not *running*, and thus the condition still holds.
   (b) Since the summary class of T is *idle*, the next two statements hold.

restart(T):

1. This is an internal action, and hence does not appear at any client automaton, and thus α' is well-formed.
2. (a) *status*(T) changes from *dead* to *not-started*, and thus this condition still holds.
   (b) *status*(T) $\neq$ *committing*, and thus this condition holds.
   (c) *status*(T) was not *committing*, and thus both sides of the equivalence were false before the action. After the action, both sides are still false.
   (d) *status*(T) was not *committed*, and thus both sides of the equivalence were false before the action. After the action, both sides are still false.
3. (a) *status*(T) was *dead*, so *status*(*parent*(T)) was and is *running*. Thus this condition holds.
   (b) *status*(T) changes from *dead* to *requested*, so the condition still holds.
   (c) No node is added to *precious*, and the value of *master* does not change, so the condition still holds.
   (d) *status*(T) = *requested*, by the postconditions of the action, so the condition holds.
   (e) The value of *precious* does not change, so the next two conditions hold because they held in the previous state.
4. (a) No node has *death-status* become *dying* or *status* stop being *running*, so the condition still holds.

5. *death-status* does not change for any node, nor does *master*. Thus the next two conditions still hold.

6. Since **restart**(T) is an internal action, the summary class of every client automaton is unchanged. Since changing the summary class of a client automaton from *dead* to *requested* does not affect any of the conditions, they still hold.

restart-precious(T):

The postconditions of this action are identical to the **restart**(T) action, except that T is added to *precious*. The preconditions are a superset. Thus, every condition not relating to *precious* holds because it held for **restart**(T).

The only conditions involving *precious* are the last four of the third statement.

3. (c) By the preconditions of the action, $master \neq T$, so the condition holds for T.

   (d) $status(T) = requested$, by the postconditions of the action, so the condition holds.

   (e) By the preconditions of the action, $parent(T) \in precious$, so $ancestors(parent(T))$ and therefore $ancestors(T) \in precious$. Thus, the condition holds.

   (f) By the preconditions of the action, no child of $parent(T)$ was a member of *precious* before the action, and hence at most one is after the action. Thus, the condition holds.

become-master(T):

1. This is an internal action, and hence does not appear at any client automaton, and thus α' is well-formed.

2. We observe that this action has no effect on the state except that the value of *master* is changed. Also, we note that it is an internal action of the deadlock recovery automaton and thus does not occur at any client automaton. Thus all conditions in this section still hold.

3. (a) This condition still holds since *status* of every node is unchanged.

   (b) $status(T) = running$, by the preconditions of the action. Thus, the condition still holds.

   (c) $T \notin precious$, by the preconditions of the action. Thus, the condition still holds.

   The rest of the conditions of this section can be easily observed not to depend on the value of *master*, and thus still hold.

4. Since $master = nil$ before this action, no node had *death-status dying* in the previous state, and thus no node does in the current state. Thus, the conditions hold.

5. The summary class of each client automaton is unchanged, as is *status* of each node. Thus, the conditions still hold.

new-master(T):

1. This is an internal action, and hence does not appear at any client automaton, and thus α' is well-formed.

2. We observe that this action has no effect on the state except that the value of *master* is changed. Also, we note that it is an internal action of the deadlock recovery automaton and thus does not occur at any client automaton. Thus all conditions in this section still hold.

3. (a) *status* is unchanged for every node, so this condition still holds.

   (b) By the preconditions of the action, *master* is a child of T, and has *status dead*. Thus, *status*(T) = *running* (by the previous condition), and the condition holds.

   (c) *master* is set to T only if $T \notin$ *precious*, so the condition holds.

   (d) Since the value of *precious* and the value of *status* for each node are unchanged, the last three conditions still hold.

4. Since *status*(*master*) was *dead*, *death-status*(*master*) was not *dying*. Thus no node has *death-status dying*, and all three conditions hold.

5. The summary class of each client automaton is unchanged, as is *status* of each node. Thus, the conditions still hold.

begin-die(T):

1. This is an internal action, and hence does not appear at any client automaton, and thus α' is well-formed.

2. We observe that this action has no effect on the state except that the value of *death-status* of one node is changed. Also, we note that it is an internal action of the deadlock recovery automaton and thus does not occur at any client automaton. Thus all conditions in this section still hold.

3. For the same reasons as for the previous section, all conditions still hold.

4. (a) By the preconditions of the action, *status*(T) = *running*, so this condition holds.

   (b) By the preconditions of the action, *death-status*(*master*) = *nil*, so before the action $\forall T$: *death-status*(T) = *nil*. Thus *death-status*(*parent*(T)) = *nil*, and since *master* = T, the condition holds.

   (c) *death-status*(*master*) = *dying*, by the postconditions, so the condition holds.

5. The summary class of each client automaton is unchanged, as is *status* of each node. Thus, the conditions still hold.

die-down(T):

1. This is an internal action, and hence does not appear at any client automaton, and thus α' is well-formed.

2. We observe that this action has no effect on the state except that the value of *death-status* of one node is changed. Also, we note that it is an internal action of the deadlock recovery automaton and thus does not occur at any client automaton. Thus all conditions in this section still hold.

3. For the same reasons as for the previous section, all conditions still hold.

4. (a) By the preconditions of the action, *status*(T) = *running*, so this condition holds.

   (b) Since in the previous state *death-status*(*parent*(T)) = *dying*, *death-status*(*master*) = *dying*, and thus *master* $\neq T$. Thus, *master*$\neq T$ in the current state, and the condition holds.

   (c) Since *death-status*(*parent*(T)) = *dying*, the consequent was true in the previous state, and is still true. Thus the condition holds.

5. The summary class of each client automaton is unchanged, as is *status* of each node. Thus, the conditions still hold.

precious(T):

1. This is an internal action, and hence dces not appear at any client automaton, and thus α' is well-formed.

2. We observe that this action has no effect on the state except that the value of *precious* is changed. Also, we note that it is an internal action of the deadlock recovery automaton and thus does not occur at any client automaton. Thus all conditions in this section still hold.

3. (a) Since *status* of every node and the value of *master* are unchanged, the first two conditions still hold.

   (c) By the preconditions of the action, $T \neq$ *master*, and the condition still holds.

   (d) Since $T \in$ *alive-children(*parent*(T))*, *status*(T) is either *requested*, *running* or *committing*, and the condition holds.

   (e) Since *parent*(T) $\in$ *precious*, *ancestors*(*parent*(T)) $\in$ *precious.* Thus, *ancestors*(T) $\in$ *precious.*

   (f) By the preconditions of the action, *children(*parent*(T))* $\cap$ *precious* = $\emptyset$, so the size of this set is at most one after one node is added to *precious.*

4. The values of *status* and *death-status* for each node are unchanged, as is the value of *master.* Thus all the conditions in this section still hold.

5. The summary class of each client automaton is unchanged, as is *status* of each node. Thus, the conditions still hold.

□

Lemma 4.4 *For any behavior of A_h,* max-resources $\leq$ *TOTAL_RESOURCES.*

Proof:

By Lemma 4.2, a client automaton occupies resources only if it is in the *started* or *child-outstanding* summary classes. By the first condition on summary classes given in Lemma 4.3, we see that *status*(T) = *running* iff the summary class is other than *idle.* Thus, a node occupies resources iff its *status* is *running.* We show by induction that in every state the number of nodes with *status running* $\leq$ TOTAL_RESOURCES.

Basis:

Observe that in the initial state no node has *status running.*

Inductive Step:

Examine all the actions, and notice that only **create** increases the number of nodes with *status running*, and only does so to one node. Examine its precondition, and see that it is required that the number of nodes with *status running* must be strictly less than TOTAL_RESOURCES, and that, therefore, the number of nodes with *status running* must be less than or equal to TOTAL_RESOURCES.

Thus, in every state the number of nodes with *status running* $\leq$ TOTAL_RESOURCES and, therefore, *max-resources* $\leq$ TOTAL_RESOURCES

□

4.3 Auxiliary properties used for liveness proof

Lemma 4.5 *The set of precious nodes contains at most one node at each level of the tree.*

Proof:

Assume this is not the case. Call the two precious nodes at the same level n_1 and n_2. Consider the lowest-level node that is an ancestor of both n_1 and n_2. It must have two precious children, one of which is an ancestor of n_1 and one of which is an ancestor of n_2. By subpoint (f) of point 3 of Lemma 4.3, this cannot be. □

Lemma 4.6 *The set of precious nodes forms a simple path from the root.*

Proof:

We know from the previous lemma that there is at most one precious node at each level. Consider the lowest-level precious node. Its parent must be precious, and so on, since the parent of every precious node is precious. Thus, the precious nodes form a simple path from the root. □

4.4 A_h meets safety part of correctness conditions

We show that every finite behavior of A_h satisfies the first five conditions for membership in *alt-beh.*

The first requirement, that the behavior is finite, is trivially satisfied.

The second requirement, that the subsequence of the behavior at each client automaton be well-formed, is satisfied, by the first point of Lemma 4.3.

The third requirement is a restriction on the amount of resources used at any one time, or, more formally, that for any behavior β of ah, *max-resources*$(\beta) \leq$ TOTAL_RESOURCES. We know this is satisfied by Lemma 4.4.

The fourth requirement states that every **create**(T) action must be preceded by a **request-create-child**(T) action with no intervening **die**(*parent*(T)). Observing the automaton code, we note that a precondition for **create**(T) is *status*(T) = *requested.* By Lemma 4.3, this implies the desired condition.

The fifth requirement states that every **commit**(T, **results**) action is preceded by a **request-commit**(T, **results**) action with no intervening **die**(T) action. Observing the automaton code, we note that the preconditions for **commit**(T, **results**) include *status*(T) = *committing* and (T, *results*) $\in$ *saved-results.* By Lemma 4.3, this implies the desired condition.

Chapter 5
Liveness Properties

This chapter presents a proof that a modified version of A_h presented here satisfies the correctness conditions, and that therefore this modified version of the algorithm is correct.

5.1 Liveness version of algorithm

We present a modified version of the deadlock recovery automaton, and show that all safety properties shown in Chapter 4 for the version of the deadlock recovery automaton presented there hold for the modified version. The modified version will be referred to as the "liveness" version; the previous version will now be referred to as the "safety" version. The composition of this modified automaton and the client automata will be referred to as A_{h_L}, and the composition of the safety version (on page 38) and the clients, previously named A_h, will henceforth be referred to as A_{h_S}.

A_{h_L} differs from A_{h_S} only in that additional preconditions have been added. From this, we will argue that all safety properties that hold for the safety version of the algorithm hold for the liveness version as well.

5.1.1 Safety properties still hold

The complete automaton description (given on page 64) is used for the proof of liveness properties. Lines in the "liveness" version that differ from those in the "safety" version are marked with a †.

For the liveness proof, we add preconditions to the actions of the automata, and only consider fair executions.

Every execution of the new automaton is an execution of the old, since "adding preconditions" has the effect of removing transitions from the automaton, because the preconditions/effects notation is merely shorthand for explicitly listing the transition function of the automaton called for by the model.

Lemma 5.1 execs(A_{h_L}) $\subseteq$ execs(A_{h_S}).

Proof:

Consider any execution of A_{h_L}. Every action taken in this execution is possible according to the definition of A_{h_S}, since it is identical, except that some actions have fewer preconditions. Thus, this execution is a execution of A_{h_S}. □

Lemma 5.2 *All properties true of all executions of A_{h_S} are true of all executions of A_{h_L}.*

Proof:

Obviously, since every execution of A_{h_L} is an execution of A_{h_S}. □

Thus, all safety properties shown for the automaton given for the safety proof will hold for the new automaton.

Lemma 5.3 fairbehs(A_{h_L}) $\subseteq$ behs(A_{h_S})

Proof:

Since the executions of the liveness version are a subset of those of the safety version, the behaviors (subsequences of schedules consisting of external actions[1]) of A_{h_L} are contained in the behaviors of A_{h_S}. Since the fair behaviors are a subset of the behaviors of a given automaton, the fair behaviors of the liveness version are contained in the behaviors of the safety version. □

5.2 Underspecified predicates

At some points in an execution of the deadlock recovery algorithm there are certain actions which could occur but are not required to occur for proper functioning. We wish in some cases to leave this choice unspecified in the high-level algorithm, leaving it to the low-level description or even to the actual implementation, to decide. Representing this in the IOA language at the high-level poses a problem, however, since if the action is enabled, it (or some other action) must eventually occur (by fairness), and if the action is not enabled it must not occur.

This underspecification allows modifications to the algorithm driven by efficiency conditions to be made after the proof is completed. For example, in this algorithm it may be found (experimentally, perhaps) desirable to kill a node only if the system is in deadlock. However, it seems that determining that the system actually is in deadlock (rather than "is probably close to") is expensive (it might require taking a global snapshot). Alternatively, experimental results might indicate that one should kill a node as soon as congestion occurs. Thus, underspecifying the conditions under which a node may be killed in the algorithm used for the proof increases the utility of the proof, since a wider range of implementations are then covered.

It may difficult be to specify the mapping between states of the low and high-levels because, for example, a low-level action may be enabled or not depending on a state component that is abstracted away. The use of underspecified predicates in the high-level automaton description allows one to specify the mapping more easily.

5.2.1 Use of underspecified predicates

In the next section we will define some underspecified predicates for use in the algorithm description for A_{h_L}. Since our definitions differ from that of a normal

[1][LT88], p. 3

predicate (i.e., a set of states in which the predicate is true), we must give meaning to algorithm descriptions written with these predicates.

Definition:

By an algorithm whose description contains a underspecified predicate, we mean the set of algorithms obtained by arbitrarily replacing each underspecified predicate with a standard predicate that meets the given conditions.

Throughout the rest of this thesis we will use A_{h_L} as shorthand for "any algorithm that is a member of the set defined by A_{h_L}." Given the definition of an underspecified predicate, we will be able to infer that, in certain states, no possible predicate that could replace the underspecified predicate could be true. Arguments about what actions *cannot* happen will be made on this basis. Similarly, we will be able to infer that, in certain states, all possible predicates that could replace the underspecified predicate must be true. Arguments about what actions *must* happen will be made on this basis.

5.3 Deadlock recovery automaton

Here, we add preconditions to many actions to make it possible to prove liveness properties for the algorithm. In A_{h_S}, many preconditions were intentionally left out to make the proof of safety properties simpler.

5.3.1 Underspecified predicates

In this section several underspecified predicates are defined. The definitions generally take the form of implications between standard predicates and a representative choice for the underspecified predicate. However, one of the critical requirements on the choice of underspecified predicates for this algorithm is that we require some underspecified predicate to be true if another is false. To express this, we must use the example member of a previously defined underspecified predicate in the definition of a succeeding underspecified predicate. This is necessary because there will be states in which two predicates may be either true or false, and we must restrict them so that it will not be the case that they are both false.

Definition:

By *example-predicate-choice*, we will refer to the standard predicate chosen to replace *example-predicate*. Obviously, it must satisfy the conditions for the underspecified predicate.

For example, in some states the underspecified predicate *create-ok* may be either true or false. However, if it is false, certain other predicates (pertaining to deadlock recovery) must be true. In addition to this scheme making the proof go nicely, it is intuitively satisfying because in writing code to implement the algorithm, the programmer may implement whatever convenient test he chooses for *create-ok*, as long as it satisfies the constraints given. Given this implementation of *create-ok*, it is trivial to implement predicates that must be true when *create-ok* is false; after *create-ok* is evaluated, then either the action it regulates should be taken, or else one of the actions pertaining to deadlock recovery.

An attempt is made here to give intuitive justifications for these definitions. This is merely to help the reader understand the motivation for them, and is in no way a proof of any properties.

We will write predicates with the argument *tree* to indicate that the predicate is a function of the entire state. We will use the parameter T to indicate that we are defining a family of predicates, parameterized on nodes of the tree.

5.3.2 Create and initiation of deadlock recovery

In general, the **create** action should be able to occur when space is available; it obviously cannot occur when there is no space. However, if the **create** action is not enabled due to lack of space, we wish some action concerned with deadlock recovery (either **become-master** or **begin-die**) to be enabled so that recovery may proceed. In fact, we may wish these situations to be mutually exclusive, in addition to being jointly exhaustive, with deadlock recovery proceeding exactly when **create** cannot occur. We wish to leave these conditions unspecified as far as is reasonably convenient.

Conditions for create

The predicate *create-ok(tree)* must be true whenever all *occupied nodes are precious* and a free node exists. Note that this does not imply that this action must immediately happen, but merely that it eventually happen or become no longer enabled.

Definition:

Let *create-ok(tree)* be an underspecified predicate such that

$$\begin{pmatrix} (\forall T : \mathit{status}(T) = \mathit{running} \Longrightarrow T \in \mathit{precious}) \wedge \\ (\|\mathit{using\text{-}resources}\| < \text{TOTAL\_RESOURCES}) \end{pmatrix} \Longrightarrow \mathit{create\text{-}ok\text{-}choice(tree)}$$

and

$$\mathit{create\text{-}ok\text{-}choice(tree)} \Longrightarrow \|\mathit{using\text{-}resources}\| < \text{TOTAL\_RESOURCES}.$$

Thus, *create-ok* must be true if all running nodes are precious, and must be false if there are no more unoccupied resources.

Observing the existing preconditions for **create**(T), we see that the precondition $\|\mathit{using\text{-}resources}\| <$ TOTAL_RESOURCES is already present. From this, it might at first seem that the second constraint *create-ok(tree)* is unnecessary, since if the existing precondition is false it matters not whether an additional one is as well. However, we also use the negation of *create-ok-choice* as a precondition for actions associated with deadlock recovery, namely **become-master** and **begin-die**. Thus, the property that *create-ok* is always false when $\|using\text{-}resources\| <$ TOTAL_RESOURCES does not hold is important.

Conditions for become-master

If there is a node T with $status(T) = requested$, but **create**(T) is not enabled, we wish **become-master**(T') to be enabled (for some T') so that deadlock recovery may begin. However, it is also acceptable for **become-master** to happen at other times.

Definition:

Let *become-master-ok*(*tree*) be an underspecified predicate such that

$$\neg create\text{-}ok\text{-}choice(tree) \Longrightarrow become\text{-}master\text{-}ok\text{-}choice(tree).$$

This ensures that if *create-ok* is false *become-master-ok* is true, but does not otherwise constrain *become-master-ok*.

We wish to allow the implementor to restrict the **become-master** action to "appropriate" nodes, without making the decision now (at proof time) of exactly what "appropriate" means. Thus, we attempt to identify the smallest set of states in which the action must be enabled for which we can reasonably easily complete the proof.

The *become-master*(T) action must be enabled if each of the following is true.

- *status*(T) is *running*.
- T has requested a child that has not been assigned space.
- no child has *status running*.

If the *become-master*(T) action is enabled in only these cases, the choice of deadlock master is confined to nodes that are leaf nodes in the tree of processes that are actually executing and for which there is some reason to believe the node will not return (having an outstanding child with no space assigned to it).

Of course, due to the inherent nondeterminism of I/O automata, an implementation is free to choose arbitrarily among all nodes for which the **become-master** action is enabled. The point here is to show that only considering nodes that meet the above conditions for deadlock mastership can still result in a correct algorithm.

Definition:

Let *master-candidate*(T, *tree*) be an underspecified predicate such that

$$\begin{array}{r} status(T) = running \wedge \\ requested - count(T) > 0 \wedge \\ \forall T' \in children(node) : status(T') \neq running \end{array} \Longrightarrow master\text{-}candidate\text{-}choice(T, tree).$$

Conditions for begin-die

Definition:
Let *begin-die-ok(T,* tree*)* be an underspecified predicate such that

$$(\textit{requested-count}(T) > 0 \wedge \neg \textit{create-ok}(\textit{tree})) \Longrightarrow \textit{begin-die-ok-choice}(T, \text{tree}).$$

Observing that it is already a precondition of **begin-die**(T) that T be the deadlock master, we see that if T has children that have been requested but not created and the **create** action is not enabled, then **begin-die**(T) must be enabled.

This is very natural from an implementation standpoint, as T may simply attempt to secure space for a child, and take the **begin-die**(T) action if this fails.

Conditions for precious

Definition:
Let *precious-ok*(T, *tree*) be an underspecified predicate, with no conditions, or more explicitly

$$\textit{false}() \Longrightarrow \textit{precious-ok-choice}(T, \text{tree}) \Longrightarrow \textit{true}().$$

Thus, *precious-ok* can be replaced with *any* predicate.

This means that an implementation of this algorithm is free to have this action happen at any time that the other preconditions are satisfied, and is also free to have it never happen.

5.3.3 Liveness version of algorithm

The following is the new algorithm with the added preconditions. Note that while some of the new preconditions use underspecified predicates, some are ordinary preconditions. New preconditions using underspecified predicates are marked with a ‡, while new ordinary preconditions are marked with †.

Input Actions:

```
    request-create-child($T$), $T$ not a root node:
      Effects:
        status($T$) := requested
    request-create-child($T$), $T$ a root node:
      Effects:
        status($T$) := requested
        precious := precious $\cup$ $T$
```

```
request-commit($T$, results):
  Effects:
    status($T$) := committing
    saved-results := saved-results $\cup$ ($T$, results)
    death-status($T$) := nil
    $\forall T' \in$ children($T$), status($T'$) := not-started

Output Actions:
  create($T$):
    Preconditions:
      status($T$) = requested
      $\|using\text{-}resources\|$ < TOTAL_RESOURCES
      create-ck(tree) ‡
    Effects:
      status($T$) := running

  commit($T'$, results):
    Preconditions:
      status($T'$) = committing
      ($T'$, results) $\in$ saved-results
    Effects:
      status($T'$) := committed
      saved-results := saved-results − ($T'$, results)
      precious := precious − $\{T'\}$
      if master = $T'$
        if parent($T'$) $\in$ precious
          master := nil
        else
          master := parent($T'$)

  die($T$):
    Preconditions:
      all children of $T$ have status $\in$ { not-started, requested, committed, dead }
      death-status($T$) := dying
    Effects:
      status($T$) := dead
      death-status($T$) := nil
      $\forall T', T' \in children(T)$, status($T'$) := not-started
```

Internal Actions:

```
restart($T$):
  Preconditions:
    $T' = parent(T)$
    $T \in$ dead-children($T'$)
    status($T'$) = running
    death-status($T'$) = $nil$
    $\neg(T' \in$ precious $\wedge$ children($T'$) $\cap$ precious $= \emptyset)$ †
    alive-children-count($T'$) = 0 †

  Effects:
    status($T$) := requested

restart-precious($T$):
  Preconditions:
    $T' = parent(T)$
    $T \in$ dead-children($T'$)
    status($T'$) = running
    death-status($T'$) = $nil$
    $T' \in$ precious $\wedge$ children($T'$) $\cap$ precious $= \emptyset$
    alive-children-count($T'$) = 0 †

  Effects:
    status($T$) := requested
    precious := precious $\cup \{T\}$

become-master($T$):
  Preconditions:
    master = $nil$
    $T \notin$ precious
    status = running
    master-candidate($T$, tree) ‡
    become-master-ok(tree) ‡

  Effects:
    master := $T$

new-master($T$):
  Preconditions:
    master $\in$ children($T$)
    status(master) = dead
    Effects:
        if $T \in$ precious then
            master := $nil$
        else
            master := $T$

begin-die($T$):
    Preconditions:
        status = $running$
        death-status($T$) = $nil$
        master = $T$
        begin-die-ok($T$, tree) ‡
    Effects:
        death-status($T$) = dying

die-down($T$):
    Preconditions:
        status($T$) = running
        death-status($parent(T)$) = dying
        death-status($T$) = $nil$
    Effects:
        death-status($T$) := dying

precious($T$):
    Preconditions:
        $T' \in$ precious
        $T \in$ alive-children($T'$)
        $T \neq$ master
        children($T'$) $\cap$ precious = $\emptyset$
        precious-ok($T$, tree) ‡
    Effects:
        precious := precious $\cup$ $\{T\}$
```

Partition of locally-controlled actions

We place all of the locally-controlled actions of the deadlock recovery automaton in one equivalence class for the purposes of fairness. Thus, any execution of the deadlock recovery automaton in which a locally-controlled action happens infinitely often is fair. Then, when we show that every fair behavior of the distributed algorithm maps to a fair behavior of the high-level automaton, we need only show that some locally-controlled action at the high level happens infinitely often. This makes showing the mapping from the low-level algorithm easier, since we must be fair to only one class.

5.4 Liveness properties

Here, we claim that all fair behaviors of A_{h_L} are contained in *alt-beh*. We use the definition of fair executions from [LT87], which is the set of executions of the result of the composition of the deadlock recovery algorithm automaton and the client automata in which an action of every class of the partition of locally-controlled actions is given a chan‹ e to occur infinitely often.

We define a metric on states of A_{h_L}, and show that every locally-controlled action decreases the metric. We then argue that the subset of metric values present in any execution is a well-founded set. We show that every fair execution is finite, and finally that every finite fair execution satisfies the correctness conditions.

5.4.1 Metric on the tree

We define a metric M on the states of the automaton A_{h_L} which maps each state to an n-tuple. The n-tuple is lexicographically ordered, e.g. (23, 459, 345) is less than (24, 1, 34) is less than (24, 2, 1).

In this section we do not attempt to prove that the metric does not increase, but merely to motivate the selection of the components. The components are listed below in order, with the first one presented being the most significant.

The components are presented in order, with the most significant being presented first. Several "components" of the metric are really several components, one for each level of the tree. This does not pose a problem, however, since the number of levels is a constant.

5.4.1.1 Number of permanently committed nodes

This component reflects the amount of "unabortable" work that has been completed. As stated in section 3.1.2 (page 35), nodes that commit to precious parents can never be killed. Thus, we refer to these nodes as *permanently committed.*

Definition:

A node T is *permanently committed* if *status*(T) is either *committing* or *committed* and *parent*(T) is precious.

We must separate the permanently committed nodes into those which are *committing* and those which are *committed* in order to make the proof work, however (because the request-commit action makes a node have *status committing* while changing the *status* of children from *committed* to *not-started*).

This component has two subcomponents for every level of the tree, with the subcomponents for the highest level (that of *super-root*) being the most significant. For each level, the higher-order subcomponent is the negative of the number of permanently committed nodes at that level with *status committed*; the lower-order subcomponent is the negative of the number of permanently committed nodes with *status committing.*

5.4.1.2 Number of precious nodes

This component is defined as the maximum possible height of the tree minus the number of precious nodes, or

$$\text{TOTAL\_RESOURCES} - |\mathit{precious}|.$$

5.4.1.3 Number of dead children of precious nodes

This component is the negative of the number of dead children of precious nodes. Recalling that *precious* is the set of precious nodes, and dead-children(n) the set of dead children of node n, we may write this component as

$$- \sum_{n \in \mathit{precious}} |\mathit{dead\text{-}children}(n)|.$$

The motivation for this is that when the number of dead children of precious nodes reaches its maximum (and therefore this component reaches its minimum), each precious node will have one child (also precious), except for the lowest-level precious node, which will have none. Then, a child of this lowest-level precious node will be restarted, and made precious.

5.4.1.4 Deadlock master is about to abdicate

This component is 1 if the deadlock master is permanently committed or is dead and its parent is precious, and 0 otherwise.

It is therefore 0 if there is no deadlock master. This component is needed because the deadlock master dying or permanently committing decreases a previous component (either the number of dead children of precious nodes or the number of permanently committed nodes), and this is not synchronous with the soon-following increase in the next (distance of deadlock master from precious process) component when the **new-master** action occurs, causing there to be no deadlock master.

5.4.1.5 Distance of deadlock master from precious process

If there is a deadlock master, the value of this component is the number of ancestors of the deadlock master (ancestors of a node includes the node) that are not precious, or

$$|\mathit{ancestors}(\mathit{master}) - \mathit{precious}|.$$

If there is no deadlock master, the value of this component is TOTAL_RESOURCES+ 1.

For example, if the parent of the deadlock master is not precious, but that node's parent is, then the value of this component of the metric is 1.

5.4.1.6 Deadlock master is alive

This component is 0 if the deadlock master is defined and has *status dead* or *committing* and is 1 otherwise. The deadlock master dying or requesting to commit decreases this, and a subsequent transfer of mastership will decrease the previous component.

5.4.1.7 Death convergecast progress

This component is designed to measure the progress of the death convergecast. In the simple case, nodes is the subtree rooted at the deadlock master have their *death-status* changed to *dying*, and later take the **die** action. This is complicated, however, by the possibility that nodes in this subtree may already be dead when the convergecast starts, and that nodes may request to commit or be committed while the convergecast is in progress. None of this detracts from the central point that the deadlock master will eventually die, but careful metric design is required in order that it may be shown that progress always does occur.

This component consists of two subcomponents for each level of the tree, with the one corresponding to the level closest to the root being the most significant. The more significant of the two subcomponents is the number of nodes at that level with a parent with *death-status dying* and *status running*. The less significant of the two subcomponents for each level is the negative of the number of nodes at that level with *death-status dying*.

The less significant of the two components at each level will decrease as the convergecast propagates down the tree, and the more significant will decrease as it propagates up. They are intertwined because the upward propagation causes the components that measure downward propagation to increase.

5.4.1.8 Number of nodes that have committed

This component is similar to the number of nodes that have permanently committed, except that there is no restriction on committing to precious nodes.

This component has two subcomponents for every level of the tree, with the subcomponents for the highest level (that of *super-root*) being the most significant. For each level, the higher-order subcomponent is the negative of the number of nodes at that level with *status committed*; the lower-order subcomponent is the negative of the number of nodes with *status committing*.

This component decreasing represents progress as long as no node dies. In this case, however, one of the higher-order components will decrease, as will be shown later.

5.4.1.9 Number of free processes

This component is the number of processes minus the number that are occupied, or the number of free processes, stated formally as

$$\text{TOTAL\_RESOURCES} - |\textit{using-resources}|.$$

5.4.1.10 Number of nodes with children needing restart

This component is the number of nodes with children that should be restarted. Let *need-restart(T)* be 1 if

$$\textit{death-status}(T) = \textit{nil} \wedge \textit{alive-children-count}(T) = 0 \wedge |\textit{dead-children}(T)| > 0.$$

Then, this component is:

$$\sum_{T \in \textit{tree}} \textit{need-restart}(T).$$

Every **restart** action decreases this component.

5.4.2 Metric never increases and always decreases

We show that no action ever increases the metric, and that every locally-controlled action decreases the metric.

Formally, if in any execution of the composition, the metric has a given value in a state, then the metric never has a higher value in a succeeding state. Further, if a locally-controlled action occurs, the metric has a lower value in the succeeding state than in the preceding state.

Lemma 5.4 *In any execution $a_0\pi_0a_1\pi_1 \ldots a_n\pi_na_{n+1}$ of the composition of the deadlock recovery automaton and client automata, the value of the metric in state a_{n+1} is not greater th.·n the value in state a_n. If π_n is a locally-controlled action of the deadlock recovery automaton, the value of the metric in a_{n+1} is less than the value in state a_n.*

Proof:

Consider each action, and for each consider all possible states. Then, argue that the action decreases or does not change the metric, the latter only for locally-controlled actions.

request-create-child(T): By examining the safety proof, we see that the *status* of T must have been *not-started.* This action changes the *status* of T to *requested*, and thus does not affect any component the metric.

request-commit(T, **results**): Before this action, the *status* of T was *running*, by the properties shown in Chapter 4. This action changes the status of T to *committing*, decreasing the component measuring the number of nodes that have committed (specifically the subcomponent for that level measuring the number of nodes at that level that have *status committing*).

If the parent of T is precious, the highest-order component is decreased, since the number of permanently committed nodes is increased.

Otherwise, if T is the deadlock master, the "deadlock master is alive" component is decreased.

Otherwise, if T has *death-status dying*, the component measuring the number of alive nodes with *death-status dying* is decreased.

Otherwise, (since T is neither precious, the deadlock master, nor has *death-status dying*), it can be seen that no components more significant than the one measuring the number of nodes that have committed change.

create(T): This action changes the status of T from *requested* to *running*. This decreases the component that measures the number of free processes, and does not affect any others, so the metric decreases.

commit(T, **results**): If the parent of T is precious, the first component of the metric decreases, since after the action T is permanently committed.

Otherwise, if T was the deadlock master, then the parent of T becomes the deadlock master, and the component measuring the distance of the deadlock master from the lowest-level precious process decreases.

Otherwise, the number of nodes that have committed increases, decreasing that component of the metric, and no other components change.

die(T): Observe that *status*(T) must be *running*, by Lemma 4.3.

If T is the deadlock master and *parent*(T) is not precious, the "deadlock master is alive" component is decreased, and no more significant components increase. If *parent*(T) is precious, then the component measuring the number of dead children of precious nodes decreases, and no more significant components increase.

If T is not the deadlock master, then the subcomponent of the convergecast component measuring the number of dead nodes with a dying parent at the level of T decreases, and no higher-order components change (although the corresponding subcomponent one level deeper increases).

restart(T): This action decreases the component measuring the number of free processes, and does not affect any higher-order components.

restart-precious(T): This action decreases the component measuring the number of precious nodes, and does not affect any higher-order components.

become-master(T): Observe that if this action occurs, then there was no deadlock master before it occurred (by the preconditions of the action). Thus, the value of the component measuring the number of non-precious ancestors of the deadlock master was TOTAL_RESOURCES + 1. After this action, it must have a lower value, since the deadlock master can have at most TOTAL_RESOURCES ancestors. No other component is affected, and thus this action always decreases the metric.

new-master(T): If there is a deadlock master after this action, it is the parent of the current one, and therefore the distance of the deadlock master from the nearest precious node has decreased, and no higher-order components change.

If there is no deadlock master after this action, then the "deadlock master is about to abdicate" component was 1 before this action, and becomes 0, and no higher-order components change.

Thus, the metric is decreased.

begin-die(T): By the preconditions of this action, T is the deadlock master and has *death-status nil.* By Lemma 4.3, no node has *death-status dying.*

This causes a node to have *death-status* dying, decreasing the subcomponent measuring the number of nodes with *death-status dying* in the component measuring convergecast progress. No higher-order components change, and the metric therefore decreases.

die-down(T): This action decreases the subcomponent for the convergecast component measuring the number of dying nodes at the level of T, and does not change any higher-order components.

precious(T): This action decreases the component of the metric that measures the number of precious nodes, and does not affect any others.

□

5.4.3 Metric is well-founded

We wish to show that there are no infinite decreasing chains in the subset of metric values corresponding to an execution of A_{h_L}. Examining the definition of the metric, we note that all components are easily observed to be finite (and bounded), except the number of permanently committed nodes, death convergecast progress, and the number of committed nodes. We can see that any node counted in one of these three components must have *status* other than *not-started*, and thus the **request-create-child** action must have occurred for that node.

At first it would seem that these components must therefore be finite, since each client automaton may only request finitely many children in a particular execution, and the depth of the tree is finite. However, consider a client automaton that requests one child when first invoked, two children when created after the first **die** action, three children the next time, and so on. Even though the client automaton requests a finite number of children in any block, the maximum number of children it requests after an infinite number of such blocks is not finite.

The strategy used here will be to show that there are no infinite decreasing chains in the first six components of the metric, and that there are no infinite decreasing chains in the rest of the metric given that the first six components do not change.

Lemma 5.5 *There are no infinite decreasing chains in first six components of metric values in the subset of metric values corresponding to an execution of A_{h_L}.*

Proof:

We examine the first six components, and show that the first component has no infinite decreasing chains, and that the others are bounded. Thus, there can be no infinite decreasing chains in the first six components of the metric.

- Number of permanently committed nodes

  Observe that every node counted by this component has a precious parent, and that therefore all ancestors are precious. Thus, no **begin-die** or **die** action can occur at any ancestor. Since *status* is *committing* or *committed*, **begin-die** cannot happen. After **commit**, *status* is *not-started*, and again **begin-die** cannot happen. Because this child has been requested already, it can never be requested again by its parent. Since the argument holds for the root node down the chain of ancestors to the node in question, no additional actions will occur at any node ever counted by this component. Thus, for all counted T, *last-block(T)* ends in **request-commit**, and will continue to end in **request-commit**, since no more actions are ever added.

  Now, consider an infinite decreasing chain, and consider the subset of nodes that were ever counted. Since the chain is infinite, there must be an infinite number of nodes in this subset (since each decrease of the metric must add a node). Next, observe that every node in the subset induced by the chain has **request-commit** as the last action. Further, we know that the parent of every such node has not performed a **die** action after any state in which the node was counted. Thus, there is a **request-create** action in the last block of the parent of every node that has ever been counted.

  We have already shown that there are an infinite number of nodes in the subset. Now, consider the tree consisting of all nodes in the above subset and all their ancestors. It must also have an infinite number of nodes, since every node in the subset is a node in this new tree. Since the induced tree has finite depth, some node must have an infinite number of children (by König's Lemma). Choose such a node arbitrarily. There must be a **request-create** in the last block for every child. But this cannot be, since it violates the definition of client automata, as client automata only request a finite number of children in any given block.

  Thus, there must not be an infinite decreasing chain.

- Number of precious nodes

  Since the set of precious nodes forms a simple path from the root (Lemma 4.6), there may be at most TOTAL_RESOURCES precious nodes. Thus this component is at least zero, and at most TOTAL_RESOURCES.

- Number of dead children of precious nodes

  There are at most TOTAL_RESOURCES precious nodes, and each has finitely many children, by client well-formedness.

- Deadlock master is about to abdicate

  This component has only two values.

- Distance of deadlock master from precious process

  The number of nodes that are ancestors of a given node is finite. Thus, this component is finite.

- Deadlock master is alive

  This component has only two values.

□

Lemma 5.6 *There are no infinite decreasing chains in the subset of metric values corresponding to an execution of A_{h_L}.*

Proof:

We have already shown in the previous lemma that there are no infinite decreasing chains in the first six components. Assuming these components do not change, we show that there are no infinite decreasing chains in the rest of the components.

Note that since one of the six higher-order components of the metric is decreased if the deadlock master dies that we may assume that the deadlock master does not die.

1. Death convergecast progress

   Observe that any node that dies either has a parent with *death-status dying* or is the deadlock master (Lemma 4.3). Examining the **restart** action, we observe that it will not be enabled on any node the parent of which has *death-status dying*. Thus, the **die** action can happen at most once for each outstanding descendant of the deadlock master. Consider the subtree of nodes rooted at the deadlock master. By the same König's Lemma argument used for permanently committed nodes, we know that there cannot be an infinite number of nodes counted by this component.

2. Number of nodes that have committed

   Given that no node dies, this component takes on finitely many values for the same reasons as for the component counting permanently committed nodes.

3. Number of free processes

   This is at most TOTAL_RESOURCES and at least zero.

4. Number of nodes with children needing restart

   Every node counted by this component has taken a **die** action, and thus this component remains finite.

Thus, since the rest of the components may take on only finitely many values for each change of the higher-order components, which we have already (in Lemma 5.5) shown may take on only finitely many values, there are no infinite decreasing chains in metric values corresponding to any execution of A_{h_L}. □

5.4.4 A_{h_L} satisfies correctness conditions

We must show that every fair behavior of A_{h_L} is contained in *correct-beh.* We do this by arguing that every execution of A_{h_L} is finite and that therefore all fair executions are finite. We then consider the final state of a fair execution execution and argue that if the liveness conditions are not satisfied that an action is enabled in the final state, and thus the execution is not fair.

Lemma 5.7 *Every execution of A_{h_L} contains a finite number of actions.*

Proof:
Since the subset of metric values possible in any execution is a well-founded set and because every locally-controlled action decreases the metric, there can only be finitely many locally-controlled actions in an execution. Thus, there are finitely many **die** actions. Since there are a finite number of client automata that take a non-zero number of steps, and each may take only finitely many steps for each **die** action, the total number of actions of client automata is finite. Since the number of both locally-controlled and input actions is finite, the total number of actions is finite. □

Lemma 5.8 *Every fair behavior of A_{h_L} contains a finite number of actions.*

Proof:
Since every fair behavior has no more actions than the corresponding fair execution, it is finite. □

Lemma 5.9 *Every fair behavior of A_{h_L} satisfies the first five conditions for membership in* alt-beh.

Proof:
By Lemma 5.3, all fair behaviors of A_{h_L} are behaviors of A_{h_S}. By Section 4.4, all behaviors of A_{h_S} satisfy these conditions. □

Theorem 5.10 *Every fair behavior of A_{h_L} is contained in* alt-beh.

Proof:
Assume there exists a fair behavior β not in *alt-beh.* Note that β is necessarily finite, by Lemma 5.8.

By the previous lemma, every fair behavior of A_{h_L} satisfies the first five conditions for membership in *alt-beh.* Because β is a fair behavior of the composition, we know that for all T, $\beta|T$ is a fair behavior of T, and thus the eighth condition for membership in *alt-beh* is satisfied. Thus, either or both of the sixth and seventh conditions must not be satisfied by β.

Consider the last state of this fair finite behavior β. Since it is fair and finite, no locally-controlled actions are enabled.

By hypothesis, we know that the one or more of the correctness conditions for A_{h_L} are not met. From this, we reach a contradiction by showing that an action is enabled.

First, we observe that certain state relationships cannot hold, since otherwise an action would be enabled.

No node may have *status committing*, since then **request-commit** for that node would be enabled.

No node T may have *death-status dying*, since then **die**(T) or **begin-die**(T'), for some child T' of T would be enabled.

If *master* $\neq$ *nil*, then *status*(*master*) = *running*, since otherwise **new-master** would be enabled.

Assume that the second condition, requiring that a **commit**(T, **results**) action follow a **request-commit**(T, **results**) action not followed by a **die**(*parent*(T)), is not met, and consider the node T for which it is not met. By Lemma 4.3, *status*(T) = *committing*, and (T, *results*) $\in$ *saved-results*. Thus, the **commit**(T, **results**) action is enabled and β is not fair.

Assume the first condition, requiring that a **create**(T) action follow a **request-create**(T) action, is not met. Since the **request-create**(T) action occurred after the most recent **die**(T) or **die**(*parent*(T)), we know that *status*(T) is *requested*, by Lemma 4.3. Examining the **create**(T) action, we find that the only other precondition is that *create-ok(tree)* be true (it includes the other condition). If this action is enabled, the execution is not fair. If it is not, then *create-ok(tree)* must be false.

If *master* has a non-*nil* value, the **begin-die**(*master*) action is enabled (since we know that *status*(*master*) = *running* and that no node has *death-status dying*).

If *master* is *nil*, then *become-master(T)* must be enabled for some T. Consider the set of nodes that occupy resources. There must be one that is not precious, for otherwise *create-ok(tree)* would be true. Since *create-ok(tree)* is false, *become-master-ok(tree)* must be true. Consider the value of *master-candidate(T,* tree*)*. If true, this action is enabled. If false, shift the focus to a child of T with *status running*. Such a child must exist because *master-candidate(T,* tree*)* was false, and it must also be non-precious. Iterating this process, we eventually come to a node with no running children, and *master-candidate* is true for this node, and hence **become-master** is enabled for this node.

□

Corollary 5.11 *Every fair behavior of* A_{h_L} *is contained in* correct-beh.

Proof:

Every fair behavior is in *alt-beh*, and *alt-beh* is a subset of *correct-beh*. □

Chapter 6
Distributed Deadlock Recovery Algorithm

In this chapter we define distributed client automata, and model the communications network and space allocation. We present the distributed version of the deadlock recovery algorithm. We prove safety properties about the distributed algorithm, and show that it solves[1] A_{h_L}, and finally that the distributed deadlock recovery algorithm is correct.

Both the communications network and the space allocation function are modeled, but we give no implementation of these functions because they are beyond the scope of this thesis. Each is assumed to be distributed in nature, and is required to have shared actions at each processor. We give constraints on those actions, and do not further constrain the implementation of these functions. The overall architecture thus consists of a number of processors, with a deadlock recovery automaton at each, all the client automata, which have shared actions at each processor, and the network and space allocation automata, which have shared actions at each processor. The space allocation automaton also shares input actions with client automata.

6.1 Modeling distributed computation

In this section we present some of the design goals of the FTPP network. We also present the naming scheme for processes used in this chapter.

6.1.1 Properties guaranteed by the FTPP network

We make several assumptions about the network communication system on which the RPC system is built. In Section 6.3, we present an automaton that models the network; this automaton is the definitive statement of the network properties. The following are some of the design goals of the communications network of the FTPP:[2]

Reliability: Exactly those messages sent are delivered, and they are delivered after being sent.

Broadcast capability: Each message sent may be sent to any subset of the set of processors.

Serialization: A consistent total ordering is imposed upon all messages sent. Messages received at each particular processor are consistent with this ordering.

[1][LT88], p. 11

[2]I was involved in several design discussions for the network. Also, see [Har85]

Bus semantics: If processor A sends message A_B to processor B, and subsequently sends message A_C to processor C, message A_B will be delivered first. Most importantly, this implies that any message C_B causally generated by C because of A_C, and then sent to B, will arrive after A_B (because A_B must already have been delivered before A_C arrived at C).

The broadcast and serialization properties imply that atomic broadcast is possible. In the FTPP hardware design, transmitted messages are actually serialized and distributed to the communications hardware at each processor; messages to fewer than all processors are simply only delivered by the communications hardware to the actual processor for the proper subset of processors.

6.1.2 Modeling of processes

As explained in Section 2.1.2, we concern ourselves with processes only, and do not address issues of supporting several processes on a single physical processor. Instead, we consider there to be one logical processor per process.

The symbol p represents a process; it ranges over the set

$$P' = \{p_i : 0 \leq i \leq \text{TOTAL\_RESOURCES}\}$$

of all processes. The set P is P' without p_0. The process p_0 is special; it is the process that evaluates *super-root*. It is not included in resource counts or broadcasts, and exists merely to make it simple to model the initial request from *super-root*.

6.2 Distributed client automata C_d

We would like the version of the client automata used with the distributed version of the algorithm, called *distributed client automata*, to be the same as the high-level client automata. However, we do not wish to clutter the high-level version of client automata with processor identifiers, since we have no notion of processors there. Thus, we form a distributed client automaton from a client automaton by adding one additional argument, a processor identifier ranging over the set of processors, to each action of every client . This models the way in which the client automata communicate with the part of the deadlock recovery algorithm running on the local processor. We could have added this argument to each action of every high-level client automaton, and not constrained its value, but instead do it here and avoid cluttering the high-level definition.

A notion of well-formedness is given for distributed client automata. This definition is similar to that for the high-level client automata, and is different only in that it enforces the notion that each "run" of a client should be on the same process.

The following is the external action signature for distributed client automata.

Input Actions:

create(T, p)

die(T, p)

commit(T', results, p), T' a child of T

Output Actions:

request-create-child(T', p), T' a child of T

request-commit(T, results, p)

6.2.0.1 Well-formedness for distributed client automata

The definition of well-formedness for distributed client automata is based on the definition for high-level client automata.

Definition:

Let M_c be a mapping from sequences of actions of distributed client automata to sequences of actions of high-level client automata. M_c applied to a sequence of one action is the action with the process specifier p removed. M_c applied to a sequence $\pi_0\pi_1...\pi_n$ is $M_c(\pi_0)M_c(\pi_1)...M_c(\pi_n)$.

Definition:

We extend the notion of "blocks" of actions defined for high-level client automata (section 2.2.3, page 21) to distributed client automata. To divide an sequence of actions of C_d into blocks, apply the mapping, divide the high-level client sequence into blocks, and then consider a block of the distributed client automaton to be those actions corresponding to a high-level block.

Definition:

A sequence of external actions α of a distributed client automaton is well-formed iff

- $M_c(\alpha)$ is a well-formed sequence of external actions of a high-level client automaton.
- The value of p for all actions in each block of α is the same.

The first condition extends the obvious properties of high-level client automata to distributed client automata. The second imposes the condition that a distributed client automaton may only communicate with the part of the deadlock recovery automaton resident on the process on which it is executing.

6.2.1 Fair behaviors of distributed client automata

Definition:

A *distributed client automaton* C_d is an automaton with the external action signature given earlier such that:

- C_d preserves well-formedness.
- For every fair behavior β of C_d, $M_c(\beta)$ is a fair behavior of some high-level client automaton C.

6.3 Communication network

We present an automaton A_N that models communication within the FTPP; it is intended to be an accurate abstraction of the actual functioning of the FTPP network. The transition function for the automaton is given below. The expression *from* represents an element of P, the set of processors. Thus, the expression q_{from} represents one of $|P|$ separate input queues, one per process. The expression *to* represents an element of the power set of P; we shall consider writing p_i, an element of P, in this place to be syntactic sugar for the singleton set $\{p_i\}$. The expression *to-proc* represents an element of P. The expression m is an element of M_{all}, the set of messages that can be sent.

The automaton code for A_N follows. For purposes of fairness, each locally-controlled action is in a separate equivalence class.

```
Input Actions:
    send(from, to, m):
      Effects:
        enqueue($q_{\text{from}}$, (from, to, m))

Output Actions:
    recv(from, to-proc, m):
      Preconditions:
        to-proc $\in$ to
        head($q_O$) = (from, to, m)
      Effects:
        to = to - { to-proc }
        if to - { to-proc } = $\emptyset$ then
          dequeue($q_O$)

Internal Actions:
    choose-proc(from):
      Preconditions:
        head($q_{\text{from}}$) = (from, to, m)
```

Effects:
  dequeue(q_{from})
  enqueue(q_O, (from, to, m))

There is one input queue for each processor; the **choose-proc** action selects a waiting message to be next in the serialized order.

One can see that in any fair behavior of this automaton:

- The actions recv(from, to-proc, m), for each *to-proc* $\in$ *to* appear iff the action send(from, to, m) does.
- If send(from, to_1, m_1) precedes send(from, to_2, m_2),

  recv(from, $to\text{-}proc_1$, m_1) precedes recv(from, $to\text{-}proc_2$, m_2),

  for $to\text{-}proc_1 \in to_1$ and $to\text{-}proc_2 \in to_2$.

Thus, all messages sent are received, no message is received that was not sent, and messages are received after they are sent. Messages sent to (possibly different) processes from a given process are received in the order sent. This realizes the "bus" design goal given in section 6.1.1.

6.4 Space allocation

When a deadlock recovery automaton has a child of the current client marked as requested, it must locate a process to execute the child. We give here correctness conditions for an automaton that solves this problem. It should be clear that a simple implementation of a space allocation automaton using a central controller could be constructed that would satisfy these conditions. However, a specification is given, rather than presenting a simple implementation, so that more efficient algorithms may be used.

The external action signature of such an automaton is given below.

Input Actions:

  create(T, p)

  die(T, p)

  request-commit(T, results, p)

  request-space(p)

Output Actions:

  can-do-work(p, p')

  no-space(p)

In order to define correctness for a space allocator, we first need a few definitions.

Definition:

Let *occupied* be a predicate on ordered pairs consisting of a sequence of external actions of client automata β and a process identifier p. The predicate has value *true* if β contains a **create** action occurring at process p that is not followed by a **die** or **request-commit** action at process p, and value *false* otherwise.

Definition:

Let *allocated* also be a predicate on ordered pairs consisting of a sequence of external actions of the space allocation automaton β and a process identifier p. Let *allocated* have value *true* if *occupied*$(\beta|p, p)$ or if β contains a **can-do-work**(p', p) action not followed by a **die** or **request-commit** action occurring at p or p', and *false* otherwise.

The predicate *allocated* can be seen to be a natural and simple model of availability for allocation. No process that is evaluating a client automaton is available, nor is any process that has been assigned if the process to which it was allocated is still occupied. This requirement allocateds the deadlock recovery algorithm from having to notify the space allocator if an allocation is not used in the event of a node being aborted.

Definition:

A space allocating automaton is an automaton such that the following are true of every fair schedule β of the automaton:

1. γ**can-do-work**$(p, p')\delta$ implies $\neg$ *allocated*(γ, p').

2. Neither **can-do-work**(p, p') nor **no-space**(p) occurs in β unless it is preceded by **request-space**(p) without any of the following actions intervening: **die**(T, p), **request-commit**$(T$, **results**, $p)$, **can-do-work**(p, p''), or **no-space**(p).

3. Every **request-space**(p) not followed by either **die**(T, p) or **request-commit**$(T$, **results**, $p)$ is followed by either **can-do-work**(p, p') or **no-space**(p).

4. If β is γ**request-space**$(p)\delta$**no-space**(p), then if $\exists p' | \neg$*allocated*(γ, p'), a **can-do-work**(p'', p''') action occurs in δ.

In other words, the space allocator only allocates processes that are not occupied and for which allocations were not previously given, and answers exactly once requests that were made and not superceded. In addition, it a request is made when there is a free process, some request is satisfied before any is denied.

6.5 Distributed algorithm

This section explains how the distributed version of the algorithm works. It is assumed that the reader already understands the high-level algorithm. As before, the normal case will be explained first, and then selection of the deadlock master and killing nodes will be explained.

In the distributed version, there is one deadlock recovery automaton per processor, and these automata communicate by sending messages using the network automaton.

In the high-level algorithm, the creation of a child is straightforward. The parent client automaton performs a **request-create-child** action, and then the deadlock recovery automaton takes a **create** action. In the distributed case, the **request-create-child** action causes the deadlock recovery automaton associated with the parent client automaton to mark the child as requested. Then, it asks the space allocation automaton for space by taking the **request-space** action. If the space allocation automaton takes the **can-do-work** action, the automaton associated with the parent sends a `<DO-WORK>` message to the specified automaton, which then takes a **create** action.

When a client automaton is finished, it takes the **request-commit** action. This causes the deadlock recovery automaton associated with that client to send a `<DONE>` message to the deadlock recovery automaton associated with the parent, which then takes the **commit** action.

When the **no-space** action occurs at a deadlock recovery automaton (under the control of the space allocator) instead of the **can-do-work** action, the deadlock recovery automaton takes action to resolve deadlock. If the deadlock recovery automaton does not have recorded that a deadlock master exists, it broadcasts a `<WANT-MASTERSHIP>` message indicating that it is a candidate for deadlock mastership. If it does have recorded that there is a deadlock master, it sends an `<AM-BLOCKED>` message to the deadlock master.

When a `<WANT-MASTERSHIP>` message is received and the local process has no deadlock master recorded, the process named in the message is recorded as the deadlock master by the receiving process, unless the receiving process originally sent the message and it is no longer eligible to become the deadlock master. Because of the properties of the communication network, we will be able to show that at most one process will think itself to be the deadlock master at any time.

When a process that is the deadlock master receives an `<AM-BLOCKED>` message, it begins to abort the subtree rooted at itself. It does this by sending `<DIE-DOWN>` messages to all of its children, which then send such messages to their children. When a node has received such a message and finds it has no children left, it executes the **die** action and then sends a `<DIE-ACK>` message to its parent. When the parent of the deadlock master receives a `<DIE-ACK>` message, it can tell that the child was the master, since the information is contained in the `<DIE-ACK>` message. If the child was the deadlock master, the parent broadcasts a `<NEW-MASTER>` message, announcing either that it is the new deadlock master, or that there is no longer a deadlock master, the latter in the case that it is precious.

6.6 State of a deadlock recovery automaton A_d

A description of the state of a distributed deadlock recovery automaton is given. Also, notation and derived variables are presented.

6.6.1 Description of the state

The following is a list of components of the state of each deadlock recovery automaton. Also given is a brief explanation of their "meaning", which is intended to help the reader follow the presentation of the algorithm code.

6.6.1.1 The variable *curT*

The variable *curT* either has value *nil* or a client automaton name (section 2.1.1). This indicates which remote procedure call is assigned to the process.

6.6.1.2 The variable *status*

The variable *status* has value *nil*, *requested* or *running.*

6.6.1.3 The variable *death-status*

The variable *death-status* has value *nil* or *dying.*

6.6.1.4 The variable *parent-process*

This variable is the process that requested the computation of the client denoted by *curT*. Its value is *nil* if $curT = nil$.

6.6.1.5 The variable *preciousp*

This variable is *true* if the process denoted by *curT* is precious, and *nil* otherwise.

6.6.1.6 The variable *work-requested*

This variable contains the identity of the remote procedure call for which the deadlock recovery automaton has selected to request space for creation. Its value is *nil* if no such request is outstanding.

6.6.1.7 The variable *master-attempted*

This variable is set to *true* if a message requesting deadlock mastership has been sent, the node is still eligible to become deadlock master, and the outcome has not been resolved; otherwise it is *nil.*

6.6.1.8 The variables *master* and *master-process*

The former is name of the client that is the deadlock master, or one of *nil* or *void*, and the latter is the process running the *master*, or *nil*. These variables will not necessarily have consistent values across the different automata.

6.6.1.9 The set *child-status*

This set contains 4-tuples for children of *curT*. Each element is of the form (*T*, *status*, *p*, *preciousp*). The elements are the child in question, its status, the process on which it is running (or *nil*), and whether it is precious. The status field for the child is similar in spirit to the *status* variable in the high-level automaton, has one of the following values:

requested: The parent of this node has requested that it be created, but no resources have yet been allocated.

running: The node is executing and has been assigned a process.

dead: The node has died.

6.6.1.10 The queues *inqueue* and *outqueue*

These queues are buffers for incoming and outgoing messages. The elements in each queue are 3-tuples of the form (*from*, *to*, *m*).

6.6.2 Initial value of the state

In the initial state of each deadlock recovery automata, the state variables have the following values:

curT:	*nil*
status:	*nil*
death-status:	*nil*
parent-process:	*nil*
preciousp:	*nil*
work-requested:	*nil*
master-attempted:	*nil*
master:	*nil*
master-process:	*nil*
child-status:	$\emptyset$
inqueue:	empty
outqueue:	empty

6.6.3 Notation for the state

Let *child-status(T)* represent the tuple element pertaining to T, or *nil* if none exists. Let *child-status(T).status* represent the *status* field, and similarly for *child-status(T).p* and *child-status(T).preciousp.*

Let *head(q)* represent the element at the head of q, or *nil* if q is empty.

Let *enqueue(q, x)* represent the operation of adding x to the tail of q.

Let *dequeue(q)* represent the operation of removing the element at the head of q.

6.7 Automaton description for A_d

6.7.1 Description of messages used

These descriptions are meant to be informative; they describe the form of messages and do not specify formal meanings, but instead attempt to provide intuition about what purposes they serve. The semantics of the messages is defined implicitly in the automaton code by what actions they cause.

<DO-WORK, *T*, **preciousp**>: This message is sent by the process running the parent of T to the process selected (by the space allocator) to evaluate T. The last component, *preciousp*, indicates that the remote procedure call should be considered precious on arrival.

<DONE, *T*, **results**, **masterp**>: This message is sent from the process evaluating T to the process running *parent(T)* to indicate that it has requested to commit with value *results*; *masterp* is *true* iff T was the deadlock master when the message was sent.

<WANT-MASTERSHIP, *T*, *p*>: This message is broadcast by p, the process evaluating T, to indicate that T is a candidate for deadlock mastership.

<NEW-MASTER, *T*, *p*>: This message is broadcast by p, the process evaluating T, to indicate that T is the new deadlock master. If T is *nil*, it indicates that there is now no master.

<AM-BLOCKED, *T*, *T'*>: This message is sent to the process of the deadlock master by a node that cannot create children when there is a deadlock master. Here, T is the blocked node and T' the deadlock master.

<DIE-DOWN, *T*>: This message is sent by T to each of its children to indicate that a death convergecast is in progress. T is specified rather than the child so that a multicast scheme could be used to send this.

<DIE-ACK, *T*, **masterp**>: This message is sent by T to its parent, indicating that it has died; *masterp* has value *true* if T was the deadlock master, and *nil* otherwise.

6.7.2 Description of actions

The following action description defines the transition relation for each distributed deadlock recovery automaton. The automaton for p_0 (which evaluates *super-root*) differs from this in that it only has the actions recv, can-do-work, send, request-work, request-create-child and commit.

Input Actions:

```
recv(from, $p$, m):
  Effects:
    enqueue(inqueue, (from, $p$, m))

can-do-work($p$, $p'$):
  Effects:
    if death-status = nil
      enqueue(outqueue, ($p$, $p'$,
        <DO-WORK, work-requested,
          child-status(work-requested).preciousp>))
      child-status(work-requested).status = running
      child-status(work-requested).$p$ = $p'$
    work-requested = nil

no-space($p$):
  Effects:
    work-requested := nil
    if no element of child-status has status = running
      if master = nil
        if preciousp = nil and master-attempted = nil
          enqueue(outqueue, ($p$, $P$, <WANT-MASTERSHIP, $T$, $p$>))
          master-attempted = true
      else if master $\neq$ nil
        enqueue(outqueue, (p, masterp,
                                  <AM-BLOCKED, curT, master>))

request-create-child($T'$, $p$), $T'$ not a root node:
  Effects:
    child-status := child-status $\cup$ ($T'$, requested, nil, nil)

request-create-child($T'$, $p$), $T'$ a root node:
  Effects:
    child-status := child-status $\cup$ ($T'$, requested, nil, true)
    request-commit($T$, results, $p$):
      *Effects*:
        if master = $curT$
          enqueue(outqueue, ($p$, *parent-process*, <DONE, $curT$, results, *true*>))
          master := void
          master-process := *nil*
        else
          enqueue(outqueue, ($p$, *parent-process*, <DONE, $curT$, results, *nil*>))
        Call procedure reset-local-state
```

Output Actions:

```
    send($p$, to, m):
      *Preconditions*:
        head(outqueue) = ($p$, to, m)
      *Effects*:
        dequeue(outqueue)

    request-space($p$):
      *Preconditions*:
        death-status = *nil*
        work-requested = *nil*
        $\exists$ ($T$, *requested*, *nil*, *preciousp*) $\in$ child-status
      *Effects*:
        work-requested := $T$

    create($T$, $p$):
      *Preconditions*:
        $curT = T$
        status := requested
      *Effects*:
        status := running

    commit($T'$, results, $p$):
      *Preconditions*:
        head(inqueue) = ($p'$, $p$, <DONE, $T'$, results, masterp>)
      *Effects*:
        dequeue(inqueue)
        child-status($T'$).*status* = *committed*
        child-status($T''$).*preciousp* = *nil*
        if masterp
          if $preciousp$
            enqueue(outqueue, ($p$, $P$, <NEW-MASTER, $nil$, $nil$>))
            master := void
            masterp := $nil$
          else
            enqueue(outqueue, ($p$, $P$, <NEW-MASTER, $curT$, $p$>))
            master := $curT$
            masterp := $p$

    die($T$, $p$):
      Preconditions:
        $T = curT$
        death-status = dying
        work-requested = $nil$
        $\forall T' \in chilren(T)$ :
            child-status($T'$).$status \in$ { not-started, requested, committed, dead }
      Effects:
        if master = $curT$
            enqueue(outqueue, ($p$, $parent\text{-}process$, <DIE-ACK, $curT$, $true$>))
            master := void
            master-process := $nil$
        else
            enqueue(outqueue, ($p$, $parent\text{-}process$, <DIE-ACK, $curT$, $nil$>))
        Call procedure reset-local-state

  Internal Actions:

    receive-do-work($T$, $p$):
      Preconditions:
        head(inqueue) = ($p'$, $p$, <DO-WORK, $T$, incoming-preciousp>)
      Effects:
        dequeue(inqueue)
        $curT := T$
        status := requested
        death-status := $nil$
        parent-process := $p'$
        preciousp := incoming-preciousp
        master-attempted := $nil$
        child-status := $\emptyset$
```

```
restart($T$, $p$):
  Preconditions:
    $curT = parent(T)$
    status = running
    death-status = $nil$
    $\exists$ ($T$, $dead$, $nil$, is-precious) $\in$ child-status
    alive-children-count(child-status) = 0
  Effects:
    child-status($T$).$status$ := $requested$
    if preciousp then
      child-status($T$).$preciousp$ := $true$
    else
      child-status($T$).$preciousp$ := $nil$

become-master($T$, $p$):
  Preconditions:
    head(inqueue) = ($p'$, $p$, <WANT-MASTERSHIP, $T$, $p'$>)
  Effects:
    if master = $nil$ then
      if $p' = p$
        if $curT = T$ $\wedge$ master-attempted = $true$
          master := $T$
          master-process := $p'$
        else
          master := void
          master-process := $p'$
          enqueue(outqueue, ($p$, $P$, <NEW-MASTER, $nil$, $nil$>))
      else
        master := $T$
        master-process := $p'$
    master-attempted := $nil$

new-master($T$, $p$):
  Preconditions:
    head(inqueue) = ($p'$, $P$, <NEW-MASTER, $T$, $p'$>)
  Effects:
    if $curT \neq$ master then
      master := $T$
      master-process := $p'$
```

```
receive-am-blocked($T$, $T'$, $p$):
   Preconditions:
      head(inqueue) = ($p'$, $p$, <AM-BLOCKED, $T$, $T'$>)
   Effects:
      dequeue(inqueue)
      if master = $T'$ = $curT$ and death-status = $nil$ then
         death-status := dying
         Call procedure do-die-down($T$, $p$)

receive-die-down($T$, $T'$, $p$):
   Preconditions:
      head(inqueue) = ($p'$, $p$, <DIE-DOWN, $T'$>)
      status $\neq$ requested
   Effects:
      if $T'$ = $parent(curT)$ then
         death-status := dying
         Call procedure do-die-down($T$, $p$)

receive-die-ack($T'$, masterp, $p$):
   Preconditions:
      head(inqueue) = ($p'$, $p$, <DIE-ACK, $T'$, masterp>)
   Effects:
      child-status($T'$).$status$ := dead
      if masterp = $true$
         if preciousp = $nil$
            enqueue(outqueue, ($p$, $P$, <NEW-MASTER, $curT$, $p$>))
            master := $curT$
            masterp := $p$
         else
            enqueue(outqueue, ($p$, $P$, <NEW-MASTER, $nil$, $nil$>))
            master := void
            masterp := $nil$
```

Procedures:

```
   reset-local-state:
         $curT$ := $nil$
         status := $nil$
         death-status := $nil$
         parent-process := $nil$
         preciousp := $nil$
```

```
        master-attempted := $nil$
        child-status := $\emptyset$

    do-die-down($T$, $p$):
        $\forall T \in$ child-status | child-status($T$).$status$ = running
          enqueue(outqueue, ($p$, child-status($T$).$p$, <DIE-DOWN, $curT$>))
```

The partition of locally-controlled actions has each in a separate class.

6.8 Automaton A_D

Definition:

Automaton A_D is the composition of the deadlock recovery automata A_d, the distributed client automata C_d, a network automaton A_N, and a space allocation automaton A_{space}.

These automata are clearly compatible, since they have no conflicting output or (by renaming if necessary) internal actions. Since no action occurs at more than two automata, they are strongly compatible.

Definition:

A deadlock recovery automaton A_d is correct if the fair behaviors of the corresponding automaton A_D are contained in *correct-beh* and A_D preserves client well-formedness.

The first part of this definition is straightforward. The last is needed to enforce the constraint that a client may be run on only one processor at a given time.

Lemma 6.1 *If A_D solves A_{h_L}, and A_D preserves client well-formedness, A_d is correct.*

Proof:

Since A_D solves A_{h_L}, the fair behaviors of A_D are contained in the fair behaviors of A_{h_L}. Since the fair behaviors of A_{h_L} are contained in *correct-beh* (Corollary 5.11), the fair behaviors of A_D are also contained in *correct-beh*. By hypothesis, A_D preserves client well-formedness, and thus A_d is correct, according to the definition above. □

6.9 Safety properties of deadlock mastership

We introduce the notion of a virtual token for deadlock mastership, argue properties of this token, and argue from this that two nodes never both think themselves to be the deadlock master.

Definition:

We say that a process p has the mastership token in a given state if *master(p)* = *curT*(p). Additionally, we say that the token exists and is in transit if one of the following holds

- a `<DONE, `T`, results, true>` message exists in any queue

- a `<DIE-ACK, T, true>` message exists in any queue

In the proof of the lemma that follows, we rely heavily on the serialization property of the communications network. Given this property, we can talk about the sequence of messages concerned with mastership.

Definition:

A message is a *deadlock message* if it is a `<WANT-MASTERSHIP>` message or if it is a `<DONE>` or `<DIE-ACK>` message with *masterp* = *true.*

Definition:

Divide the sequence of messages delivered by the communications network automaton into rounds (and messages not part of any round) as follows. All messages before the first deadlock message are not part of any round. The first deadlock message and all succeeding messages up to and including a `<NEW-MASTER(`*nil*, *nil*`)>` message constitute a round. Messages after this `<NEW-MASTER>` message up to but not including the next deadlock message are not part of a round. The next deadlock message begins the next round, and so on.

Lemma 6.2 *In any state of an execution of A_D, either no process has the token and no token is in transit, one process has the token, or one token is in transit.*

Proof:

The proof is by induction on the length of executions.

Basis:

In the initial state, no process has the token because *master* is everywhere *nil.* There are no messages in any of the queues, so no tokens are in transit.

Inductive Step:

From examining the automaton code, we see that only the following actions can possibly affect a token: **request-commit, commit, die, receive-die-ack, new-master, become-master.**

request-commit(T, results, p): If before this action p does not have the token, then after this state it still does not and no token is introduced into transit. If before this action p has the token, then after this action it does not (because *master*(p) is set to *nil*), and one token is introduced into transit. Thus, the condition still holds.

commit(T, results, p): If the `<DONE>` message removed from the input queue of p by this action does not have *masterp true*, no change in token ownership of tokens in transit occurs. If the message has *masterp true*, then there is a token in transit in the previous state, and thus no process has the token in that state. After the action, there is no token in transit and p may have the token, or may not (depending on the value of *precious* at p). Thus, the condition holds after this action.

die(T, p): If p does not have the token in the state before this action, the <DIE-ACK> message placed into p's output queue does not have *masterp true*, and no token is generated. If p does have the token in the previous state, then no token can be in transit in that state. Then, the *masterp* component of the <DIE-ACK> message has value *true*, and there is one token in transit after the action. Also, *master* at p is set to *nil*, so p does not have the token in the state after this action.

receive-die-ack(T, p): If the <DIE-ACK> message removed from the input queue of p by this action does not have *masterp true*, no change in token ownership of tokens in transit occurs. If the message has *masterp true*, then there is a token in transit in the previous state, and thus no process has the token in that state. After the action, there is no token in transit and p may have the token, or may not (depending on the value of *precious* at p). Thus, the condition holds after this action.

new-master(T, p): Examining the automaton code for this action, we see that it can never cause p to have the token because it does not change the state at all if $curT = T$.

become-master(T, p): If the message at the head of p's input queue is not <**WANT-MASTERSHIP**, T, p>, p does not acquire a token and none is placed in transit.

Otherwise, if this action begins a round, then the preceding message in the global order pertaining to deadlock mastership is <**NEW-MASTER**, *nil*, *nil*>. The deadlock recovery automaton that sent this message set *master* (locally) to *void* when it was sent, and had the token at the time (by examination of the deadlock recovery automaton code). Thus, just after this message was sent, no process had the token and none were in transit. Thus, just before the **become-master** action this is still the case (because no action except **become-master** can cause a token to exist where none did previously). By examining the code for **become-master**, we see that only one process can possibly acquire a token in processing a given message (because the process identifier in the message must match that of the process, and process identifiers are unique). Thus at most one process has a token after this action in this case.

Otherwise, if this action does not begin a round, then it is in a round, and there is a <**WANT-MASTERSHIP**> message before the current message in the global ordering which begins this round, and that message has already been received by p. We examine the code for **become-master**, and see that if *master* was *nil* when this message was processed, then *master* was set to a non-*nil* value. Examining the rest of the code, we observe that *master* is never set to *nil* except on receiving a <**NEW-MASTER**, *nil*, *nil*> message. But no such message is part of the global message ordering after the <**WANT-MASTERSHIP**> message that begins this round, because otherwise the message beginning this round and the <**WANT-MASTERSHIP**> message at the head of p's input queue would be in different rounds. Thus, *master* is still non-*nil*, and its value is not changed. Thus, p does not acquire a token by this action and the condition holds if it held in the previous state.

□

6.10 A_D implements A_{h_L}

In this section, we show that the low-level automaton *implements* the high-level one, which means that the finite behaviors of A_D are contained in the finite behaviors of A_{h_L}.[3] To do this, we give a possibilities mapping[4] from A_D to A_{h_L}.

6.10.1 Possibilities mapping

A possibilities mapping S consists of a mapping of states of the low-level to the power set of states of the high-level, such that the following conditions hold[5]:

1. For every start state s_0 of A_D, there is a start state t_0 of A_{h_L} such that $t_0 \in S(s_0)$.
2. If s' is a reachable state of A_D, $t' \in S(s')$ is a reachable state of A_{h_L}, and (s', π, s) is a step of A_D, then there is an extended step[6] (t', γ, t) of A_{h_L} such that
   (a) $\gamma|\text{ext}(A_{h_L}) = \pi|\text{ext}(A_D)$, and
   (b) $t \in S(s)$.

We, however, will only use s and $s'\pi s$ (no actions or a single action) as extended steps.

In giving possibilities mapping, we are required to exhibit a state mapping, and prove the existence of appropriate extended steps. We present a mapping from states of A_D to states of A_{h_L} (rather than to the power set of states of A_{h_L}); this simply allows us to write $t = S(s)$ rather than $t \in S(s)$.

We then present a mapping of pairs of states of A_D and actions of A_D to either an action of A_{h_L} or the null action. This allows us to prove the existence of the required extended steps. This method is inspired by the definition of abstraction mappings in [WLL88], p. 111.

6.10.1.1 State mapping

We present a mapping from states of A_D to states of A_{h_L}.

To obtain the state of A_{h_L} corresponding to a state of A_D, apply the rules below. In cases of conflict, the higher-numbered rule prevails.

[3][LT88], p. 11

[4][LT88], p. 15

[5]This definition is taken from [LT88], p. 15, with A_D substituted for the low-level automaton, and A_{h_L} for the high-level.

[6]On page 15 of [LT88] Lynch and Tuttle define an extended step of an automaton A as "a triple of the form (s', β, s), where s' and s are steps of A, β is a finite sequence of actions of A, and there is an execution fragment of A having s' as its first state, s as its last state, and β as its schedule." They note that the "execution fragment might consist of a single state, in the case that β is the empty sequence."

1. *status*(T) = *not-started*

2. *status*(T) = *committing* and $(T$, *results*$)$ $\in$ *saved-results* if and only if a `<DONE,` T`, results, masterp>` message exists in any queue.

3. *status*(T) = *committed* if $\exists A_d$ with *curT* = *parent*(T) and *child-status*(T).*status* = *committed*

4. *status*(T) = *requested* if one of the following hold

   - $\exists A_d$ with *curT* = *parent*(T) and *child-status*(T).*status* = *requested*
   - a `<DO-WORK,` T`,` *preciousp*`>` exists in any queue
   - $\exists A_d$ with *curT* = T and *status* = *requested*

5. *status*(T) = *running* iff $\exists A_d$ with *curT* = T and *status* = *running*

6. *status*(T) = *dead* if either of the following hold

   - a `<DIE-ACK,` T`, masterp>` exists in any queue.
   - $\exists A_d$ with *curT* = *parent*(T) and *child-status*(T).*status* = *dead.*

7. $T \in$ *precious* iff either of the following hold

   - $\exists A_d$ with *curT* = T and *preciousp* = *true*
   - $\exists A_d$ with *curT* = *parent*$(T)$and *child-status*(T).*preciousp* = *true*

8. *master* = *nil*

9. *master* = T if a `<DIE-ACK,` T`,` *true*`>` or `<DONE,` T`, results,` *true*`>` message exists in any queue

10. *master* = T if $\exists A_d$ with *master* = T = *curT*

11. *death-status*(T) = *nil*

12. *death-status*(T) = *dying* if $\exists A_d$ with *curT* = $T \wedge$ *death-status* = *dying*

6.10.1.2 Action mapping

Here, we present a mapping from pairs (s, π) of states and actions of A_D to actions of A_{h_L} or the empty sequence. Since some actions map to the empty sequence regardless of the state, the state is only mentioned when it matters. In other cases, the states that produce the various outcomes are listed.

recv, can-do-work, no-space: These action maps to the empty sequence.

request-create-child(T, p): This maps to **request-create-child(T)**.

request-commit(T, results, p): This maps to **request-commit(T, results)**.

send: This maps to the empty sequence.

request-space: This maps to the empty sequence.

create(T, p): This maps to **create(T)**.

commit(T', results, p): This maps to **commit(T', results)**.

die(T, p): This maps to **die(T)**.

receive-do-work: This maps to the empty sequence.

restart(T, p): This maps to **restart(T)** or **restart-precious(T)**. The latter is chosen whenever *parent*(T) is precious.

become-master(T, p): If *master*(p) = *nil*, the processor named in the message at the head of p's input queue is p, *master-attempted*(p) is *true*, and *curT*(p) = T, this maps to **become-master(T)**. Otherwise it maps to the empty sequence.

new-master(T, p): This maps to the empty sequence.

receive-am-blocked(T, T', p): If *curT*(p) = T', *death-status*(p) = *nil* and *master*(p) = T', this maps to **begin-die(T')**, and the empty sequence otherwise. In other words, if the processor receiving the message has T', is the deadlock master and has not already taken this action, then the low-level action maps to **begin-die**.

receive-die-down(T, T', p): If *parent*(*curT*(p)) = T', this maps to **die-down(T)**, and otherwise to the empty sequence. In other words, for the output to be non-empty, T' is the parent of the process running on p.

receive-die-ack(T', masterp, p): If *masterp* is *true* and *preciousp*(p) is *nil*, this maps to **new-master(*parent*(T'))**. If *masterp* is *true* and *preciousp*(p) is *true*, this maps to **new-master(*nil*)**. Otherwise, it maps to the empty sequence.

6.10.2 Safety properties of the low-level algorithm

We must show several safety properties of A_D in order to show that the action mapping is valid.

Lemma 6.3 death-status*(p)* = dying *if there exists a (*T', running, p', preciousp*) in* child-status*(p) for which the first statement holds, and* death-status*(p)* = dying *only if for each (*T', running, p', preciousp*) in* child-status*(p) exactly one of the following hold:*

1. *One (p, p′,* `<DIE-DOWN,` *parent(T′)>) tuple exists in some queue.*
2. death-status*(p′)* = dying *and* curT*(p′)* = T'.
3. *One tuple (p′, p,* `<DIE-ACK,` T'`, masterp>`*) exists in some queue.*
4. *One tuple (p′, p,* `<DONE,` T'`, results, masterp>`*) exists in some queue.*

Proof:

The proof is by induction on the length of executions.

Basis:

Examine the shortest execution, consisting of only the start state, and observe that all queues are empty and that no deadlock recovery automaton has *death-status dying*, and that therefore the antecedent is false and the statement holds.

Inductive Step:

Given an execution for which the property holds, we show that each of the executions formed by adding an action satisfy the property. We do this by enumerating the actions. Actions for which the argument is trivial are omitted.

recv(from, p, m): This action merely moves tuples between queues, and thus the statement is still true.

request-commit(T, results, p): After this action, *child-status*(p) is $\emptyset$, and *death-status*(p) is *nil*. Thus, the statement holds.

send(p, to, m): This action merely moves tuples between queues, and thus the statement is still true.

commit(T', results, p): If *death-status*(p) is not *dying* before this action, then it is not so after the action, and the statement holds. If *death-status*(p) = *dying* before this action, then the fourth statement (that (p', p, `<DONE,` T'`, results, masterp>`) exists in some queue) holds for T', and none of the others do. After the action, *child-status*(T')*.status* is *committed*, and the *child-status* entry for T' is no longer relevant since it is not selected by the universal quantifier.

die(T, p): After this action, *child-status*(p) is $\emptyset$, and *death-status*(p) is *nil*. Thus, the statement holds.

receive-am-blocked(T, T', p): By the postconditions of this action, for every T' for which *child-status*(T').*status* = *running*, there is a (p, p', `<DIE-DOWN,` T>) tuple in the output queue of p. Also, *death-status*(p) = *dying*. Thus, the statement holds after this action.

receive-die-down(T, T', p): If $T' \neq$ *parent*(*curT*), the condition holds after the action because the state is unchanged. Since the first condition of the consequent of the lemma held before the action, the second did not (because *death-status*(p) is *dying* only if exactly one of the four conditions hold). Thus, *death-status*(p) was *nil* before this action. Thus, no `<DIE-DOWN,` *curT*(p)> messages existed in any queues before this action. After the action, one such message exists for every child of *curT*(p) = T, and *death-status*(p) = *dying*.

Otherwise, there is no change in any of the components of the state.

receive-die-ack(T', *masterp*, p): If *death-status*(p) is not *dying* before this action, then it is not so after the action, and the statement holds. If *death-status*(p) = *dying* before this action, then the third statement (that (p', p, `<DIE-ACK,` T'`, masterp>`) exists in some queue) holds for T', and none of the others do. After the action, *child-status*(T').*status* is *dead*, and the *child-status* entry for T' is no longer relevant since it is not selected by the universal quantifier.

□

Lemma 6.4 *For any execution α of A_D, if in the last state* child-status*(T) at p* $\neq$ nil, *then* curT*(p) is* parent*(T) and* status*(p) is* running.

Proof:

The proof is by induction on the length of executions.

Basis:

Examine the shortest execution, consisting of only the start state, and observe that *child-status* is everywhere the empty set, and thus the statement holds.

Inductive Step:

Given an execution A for which the property holds, we show that each of the executions $A\pi$ formed by adding an action satisfy the property. We do this by enumerating the actions, omitting the argument for actions which obviously or trivially preserve the statement.

request-create-child(T', p): By client automata well-formedness, this action can only occur if the **create**(T, p) action, T = *parent*(T'), occurs in *restricted-last-blocks*(A). Thus, *curT*(p) is *parent*(T'), and the condition holds.

request-commit(T, **results**, p): This action sets *child-status* to the empty set, so the condition holds after it.

die(T, p): This action sets *child-status* to the empty set, so the condition holds after it.

receive-do-work(T, p): This action sets *child-status* to the empty set, so the condition holds after it.

□

We present three claims. Unfortunately to prove that each claim holds for a given state, we need to assume not only that that claim holds in the previous state, but that the others do as well. So, instead of giving an inductive proof for each claim, we show instead that

- The claim holds in the initial state.
- If this claim **and the other two claims** hold in state n, this claim holds in state $n + 1$.

Then, we argue that the collection of these three proofs simultaneously prove the validity of the three claims.

Claim 6.5 *For any execution α of deadlock recovery automaton p, in the last state,* child-status*(T')*.status = running *if and only if in the last state exactly one of the following hold:*

1. *a (p, p′,* `<DO-WORK,` T'`, is-precious>`*) tuple exists in any queue,*
2. *some deadlock recovery automaton p′ has* curT = T', parent-process = *p, and* status ≠ nil,
3. *a (p′, p,* `<DONE,` T'`, results, masterp>`*) tuple exists in any queue, and*
4. *a (p′, p,* `<DIE-ACK,` T'`, masterp>`*) tuple exists in any queue,*

Proof:
The proof is by induction on the length of executions.

Basis:
Examine the shortest execution, consisting of only the start state, and observe that each of the branches of the antecedent is false, and thus the statement holds.

Inductive Step:
Given an execution for which the property holds, we show that each of the executions formed by adding an action satisfy the property. We do this by enumerating the actions.

recv(from, p, m): This moves messages between queues, and does not affect any conditions of the statement.

can-do-work(p, p'): If *death-status*(p) is *dying*, this action has no effect and the statement still holds after it. By the definition of a space allocation automaton, there was a request-space(p) action in α not followed by any of the following actions: die(T, p), request-commit(T, results, p), can-do-work(p, p''), or no-space(p). By Claim 6.6 *child-status*(*work-requested*)*.status* at p was *requested*, and was thus not *running*. Therefore, none of the four conditions held in the previous state, and thus exactly one holds after this action if a message is enqueued by this action and the appropriate *status* field of *child-status* is set to *running*. If no message is enqueued, then the statement still holds because it held in the previous state.

no-space(p): The statement holds because it held in the previous state.

request-create-child(T', p): The statement holds because it held in the previous state.

request-commit(T, **results**, p): By client well-formedness, a **create**(T, p) action previously occurred. Thus, *status*(p) is non-*nil*. Thus, by the second statement of the lemma, *child-status*(T) at *parent-process*(p) is *running*. Examining the effects of the action, we see that after the action *status*(p) is *nil*, and that the `<DONE>` message referenced in the third statement is enqueued.

send(p, **to**, **m**): The statement still holds because this action only moves messages between queues.

request-space(p): The statement still holds because the only change in the state is the value of *work-requested*, and this variable does not appear in the statement.

create(T, p): The statement still holds because the only change in the state is the value of *status*, and it is changed from *requested* to *running*, neither of which is *nil*.

commit(T', **results**, p): In the state before this action, there was a `<DONE,` T'`, results, masterp>` message in the input queue of p, and thus *child-status*(T').*status* is *running*, and none of the other 3 subconditions hold (because exactly one must hold if *child-status*(T').*status* is *running*). In the state after this action, none of the four subconditions hold, and *child-status*(T').*status* $\neq$ *running*. Thus, the statement still holds.

die(T, p): Observing the effects of this action, we can see that a `<DIE-ACK,` *curT*`, masterp>` message is sent to the process *parent-process*, and the values of *curT* and *parent-process* are reset to *nil*. Thus, for the process *parent-process*, the fourth subcondition is substituted for the second, and for any other process, the state of any subcondition is unchanged. Also, observe that a precondition is that no child has a recorded state *running*, so the resetting of the local state (which sets *child-status* to $\emptyset$) does not change the state of a child to or from *running*.

receive-do-work(T, p): This action removes the `<DO-WORK,` T`, incoming-preciousp>` message from the input queue of p, and sets *curT* to T, *parent-process* to the process sending the message, and *status* to *requested*. Thus, the property still holds for *parent-process*.

By Claim 6.7, we know that p is unoccupied and allocated. Thus, no **create**(T, p) action has occurred that is not followed by **request-commit**(T, **results**, p) or **die**(T, p), and by examining the initial state and the postconditions of **request-commit** and **die**, we see that *curT*(p) is *nil* in the previous state. Thus, changing it does not cause the statement to be violated.

restart(T, p): This action merely changes the status of T as recorded at *parent*(T) from *dead* to *requested*, so the condition holds after the action because it holds before.

become-master(T, p): This action does not enter into or remove from any queue any message referenced in the statement, and does not change the value of any referenced state components. Thus, the condition still holds.

new-master(T, p): The condition still holds, for the same reasons as the previous action.

receive-am-blocked(T, T', p): The condition still holds, for the same reasons as the previous action.

receive-die-down(T, T', p): The condition still holds, for the same reasons as the previous action.

receive-die-ack(T', masterp, p): Before this action, there is a <`DIE-ACK`, T', **`masterp`**> action in the input queue of p, so *child-status*(T')*.status* = *running* at p. Thus, none of other four subconditions holds in the state before the action, since exactly one must hold. After this action, *child-status*(T')*.status* is *dead*, and none of the four subconditions hold (since the message is removed from the queue). Thus, the statement still holds.

□

Claim 6.6 *For any execution α of deadlock recovery automaton p, the last state of α has* child-status(work-requested).status = requested *if there is a* request-space(p) *action in α not followed by any of the following actions:* die(T, p), request-commit(T, results, p), can-do-work(p, p''), *or* no-space(p).

Proof:
The proof is by induction on the length of executions.

Basis:
Examine the shortest execution, consisting of only the start state, and observe that there is no request-space(p) action, and therefore that the antecedent is false and the statement holds.

Inductive Step:
Given an execution for which the property holds, we show that each of the executions formed by adding an action satisfy the property. We do this by enumerating the actions.

First, we observe that many actions are not mentioned in the statement of the property and that therefore their occurrence in and of itself can not cause the statement to become false. Also, the effects clauses of many of these actions do not alter the value of *child-status*, and thus also may not cause the statement to become false. The following actions fall into this category:

recv(from, p, m):

send(p, to, m):

create(T, p):

receive-do-work(T, p):

become-master(T, p):

new-master(T, p):

rece[illegible]-am-blocked(T, T', p):

receive-die-down(T, T', p):

We enumerate the other actions, and show that an execution formed by adding each to an execution satisfying the statement satisfies the statement.

can-do-work(p, p'): After this action, the antecedent is false, so the condition holds.

no-space(p): After this action, the antecedent is false, so the condition holds.

request-create-child(T', p): After this action, *child-status*(T')*.status* is *requested*, and thus the condition is satisfied if *work-requested* $= T'$. If *work-requested* has some other value, the condition is satisfied because it held in the previous state.

request-commit(T, **results**, p): After this action, the antecedent is false, so the condition holds.

request-space(p): After this action, the consequent of the statement holds.

commit(T', **results**, p): *child-status*(T')*.status* is *running*, since a `<DONE,` T'`, results, masterp>` message exists in the input queue, by Claim 6.5. Thus if the **request-space** action has occurred and not been canceled as specified in the statement of the lemma, *child-status*(T'')*.status* is *requested* for some other child T'', and still has this value after the action, and the condition still holds.

die(T, p): After this action, the antecedent is false, so the condition holds.

restart(T, p): Since the condition was true before and *child-status*(T)*.status* was *dead*, *work-requested* could not have been T. Thus, the condition still holds.

receive-die-ack(T', **masterp**, p): *child-status*(T')*.status* is *running*, since a `<DIE-ACK,` T'`, masterp>` message exists in the input queue, by Claim 6.5. Thus if the **request-space** action has occurred and not been canceled as specified in the statement of the lemma, *child-status*(*work-requested*)*.status* is *requested* and thus *work-requested* $\neq T'$, and thus condition still holds.

□

Claim 6.7 *For any execution α of A_D, in the last state the following are true:*

1. *There is at most one (p', p,* `<DO-WORK,` T`, preciousp>`*) tuple in any queue.*
2. *If deadlock recovery automaton p has* status requested, *no such tuple exists in any queue.*
3. *A (p', p,* `<DO-WORK,` T`, preciousp>`*) tuple exists in some queue or deadlock recovery automaton p has* status requested *if and only if process p is unoccupied and allocated.*

Proof:

The proof is by induction on the length of executions.

Basis:

In the initial state, all queues are empty, *status* is *nil* at every deadlock recovery automaton, and no process is allocated. Thus, the statement holds.

Inductive Step:

Given an execution for which the property holds, we show that each of the executions formed by adding an action satisfy the property. We do this by enumerating the actions. Many actions for which the claim is that the statement holds after because it held before and no effect of the action changed any state components listed in the condition are omitted.

recv(from, p, m): This action merely moves messages between queues, and thus the statement still holds.

can-do-work(p', p): By the first condition on schedules of the space allocation automaton, p is unallocated (and thus unoccupied) in the previous state. Thus no appropriate tuple exists in the previous state, nor does *status*(p) = *requested*, since the statement was true in the previous state. After this action, p is allocated, but still unoccupied, by inspection of the definitions of allocated and occupied. Also, a (p, p', <`DO-WORK`, T, *preciousp*>) tuple is added to the output queue of deadlock recovery automaton p. Since there was no such tuple before, there is exactly one after. Thus, the statement holds after this action.

request-commit(T, results, p): Before this action, p was occupied (by client well-formedness, a create(T, p) action must exists in *last-block*(α)), and therefore allocated. Because the statement held in the previous state, no appropriate tuple existed in any queue, and p did not have *status requested*. After the action, p is not occupied, and not allocated, by the definitions of occupied and allocated. Since no appropriate tuple exists in any queue (because it did not in the previous state), and *status*(p) $\neq$ *requested*(by the postconditions of the action), the statement holds after the action.

send(p, to, m): This action merely moves messages between queues, and thus the statement still holds.

create(T, p): Before this action, *status*(p) = *requested*. Thus, no appropriate tuple exists in any queue. After the action p is occupied (therefore allocated). After the action *status*(p) = *running*, by the postconditions of the action, and no appropriate tuple exists in any queue, because it did not before. Thus, the statement holds in the state after the action.

die(T, p): Before this action, p is occupied and allocated, and after it is unoccupied and unallocated. Thus, no (p', p, <`DO-WORK`, T, `preciousp`>) tuple exists in any queue nor does p have *status requested* in either the state before or after this action.

We must also verify that these conditions hold for all children of client automaton being run on p. The first two hold because they held in the previous state.

By the preconditions of this action, *child-status*(T').*status* $\neq$ *running*. By Claim 6.5, no $(p, p'$, `<DO-WORK, T', preciousp>`) tuple exists in any queue, for any p' and T' a child of T. Thus, there does not exist in the state before this action a child process that is unoccupied and allocated. Thus, after the action there does not exist a child process that is unoccupied and allocated, by the definition of allocated.

receive-do-work(T, p): Before this action, a $(p', p$, `<DO-WORK, T, preciousp>`) message exists in the input queue of deadlock recovery automaton p, by the precondition of the action. Thus, there is no other appropriate tuple in any queue. After the action, *status*(p) = *requested* and the above tuple no longer exists. As before, p is unoccupied and allocated. Thus, the condition still holds.

□

Lemma 6.8 *Claims 6.5, 6.6, 6.7 are true.*

Proof:

The proof is by induction on the length of executions.

Basis:

We already showed that each claim holds in the initial state.

Inductive Step:

Assume that all claims hold in a given state. By the proofs given with each claim, each claim holds in the next state.

□

Lemma 6.9 *In the last state of an execution,* curT $\neq$ nil *iff* status $\neq$ nil.

Proof:

The proof is by induction on the length of executions.

Basis:

Examine the shortest execution, consisting of only the start state, and observe that everywhere *curT* and *status* both have the value *nil.*

Inductive Step:

Given an executions satisfying the property, consider adding each action to the existing execution to form a new execution.

request-commit$(T$, results, $p)$: After this action *status*(p) is *nil*, as is *curT*(p). Thus the condition holds after this action.

create(T, p): By the preconditions of this action, *status*(p) is non-*nil*, and thus *curT*(p) is non-*nil*, by the inductive hypothesis. Thus, after the action *curT*(p) is non-*nil*, and *status*(p) is *running*. Thus the condition holds after this action.

die(T, p): After this action *status*(p) is *nil*, as is *curT*(p). Thus the condition holds after this action.

receive-do-work(T, p): After this action, *status*(p) is *requested*, and *curT* contains the value that was in the message. Examining the automaton code, we see that only the can-do-work action creates `<DO-WORK>` messages, and by Claim 6.6, the second component of the message (the one assigned to *curT*) is never *nil*.

□

Lemma 6.10 *For any execution* α *of* A_D*, if in the last state* master*(p)* = nil*, and* master-attempted*(p)* = true*, then* preciousp*(T) is* nil *and* status*(p) is* running.

Proof:

The proof is by induction on the length of executions.

Basis:

Examine the shortest execution, consisting of only the start state, and observe that *master-attempted* is everywhere false. Therefore, the statement holds.

Inductive Step:

Given an execution for which the property holds, we show that each of the executions formed by adding an action satisfy the property. We do this by enumerating the actions. Actions for which the argument is trivial are omitted.

no-space(p): Since no-space occurs, *child-status*(*work-requested*).*status* = *requested*, by Claim 6.6. Thus, *status*(p) is *running*, by Lemma 6.4, and the statement is true after this action.

request-commit(T, results, p): By the postconditions of the action, the antecedent of the statement is false, and thus the statement holds.

create(T, p): Since *status* is *requested* in the previous state, the antecedent must have be false. Thus it must be false in this state as well, since neither *preciousp* nor *master-attempted* is changed.

die(T, p): By the postconditions of the action, the antecedent of the statement is false, and thus the statement holds.

receive-do-work(T, p): By the postconditions of the action, the antecedent of the statement is false, and thus the statement holds.

become-master(T, p): The value of *master-attempted* is *nil* after this action, so the statement holds.

□

6.10.3 S is a possibilities mapping

Here we show, using the action mapping defined in Section 6.10.1.2, that S is a possibilities mapping from A_D to A_{h_L}.

Lemma 6.11 *S applied to every start state of A_D yields a state state of A_{h_L}.*

Proof:

Consider s, the sole start state of A_D. Clearly $P(s)$, since s is reachable. Finding $S(s)$, as described in section 6.10.1.1, we see that it is exactly the start state of A_{h_L}. □

The definition of possibilities mapping requires that an extended step of A_{h_L} exist for each step of A_D. For any step (s', π, s) of A_D, we claim that $(S(s'), A(s', \pi), S(s))$ is a step of A_{h_L} (or else merely a state of A_{h_L}, in the case when $A(s\,\pi)$ is null).

Lemma 6.12 *For any (s', π, s) a step of A_D, $\pi|\text{ext}(A_D) = A(s', \pi)|\text{ext}(A_{h_L})$.*

It is easily observed that each external action of A_D maps into the same action of A_{h_L} (dropping the p argument) and that no other action of A_D has such actions in the range of the mapping function.

Lemma 6.13 *If s' is a reachable state of A_D and (s', π, s) is a step of A_D, then $s'A(s', \pi)s$ is an extended step of A_{h_L}.*

Proof:

We enumerate the actions of A_D, and consider (s', π, s) for arbitrary s' and s.

recv(from, p, m): This action maps to the empty sequence. Note that it merely moves messages between queues, and thus does not change the high-level state.

can-do-work(p, p'): This action maps to the empty sequence. This action causes the recorded state of one of the children of T, the node at p to be changed from *requested* to *running*, and a <DO-WORK, T', *preciousp*> message to be enqueued. On examining the state mapping, one sees that *status*(T') remains *requested*, since either the state being recorded as *requested* at the parent or the existence of such a message suffices to make the high-level state *requested.*

no-space(p): This action maps to the empty sequence. It causes no high-level state change, as can be seen from examining the state mapping and noting that neither of the two messages this action can enqueue are mentioned.

request-create-child(T, p): This maps to request-create-child(T), which is an input action and therefore always enabled. Consider the difference between s' and s. The only change is that a new node has been added to the *child-status* field of the process with *parent*(T), with *status requested.* Examining the state mapping, one sees that this causes $S(s)$ to also change by T having *status requested.* Examining the code for A_{h_L}, one sees that the effect of the request-create-child(T) action is to change the *status* of T to *requested.*

request-commit(T, **results**, p): This maps to **request-commit**(T, **results**), which is an input action and therefore always enabled. The action of A_D modifies the state by enqueueing a <DONE> message. By examining the state mapping, one can see that *status*(T) of A_{h_L} changes to *committing*, just as the **request-commit** action of A_{h_L} requires. The state of any children changes to *not-started*, as the high-level action requires. The (T, results) pair is added to *saved-results* as required because the state mapping stipulates that such a pair belongs to *saved-results* if an appropriate <DONE> message exists. We also observe that if the node that requests to commit is the deadlock master, the <DONE> message encodes this, and the existence of this message causes the value of the deadlock master at the high level to be unchanged. Thus, $S(s)$ is the state of A_{h_L} after this action.

send(p, **to**, **m**): This maps to the empty sequence. It merely moves a message between queues, and no part of the state mapping is affected by this. Thus, the high-level state is unchanged.

request-space(p): This maps to the empty sequence. The value of *work-requested* (the only state component changed) does not affect the state mapping. Thus, the high-level state is unchanged.

create(T, p): This maps to **create**(T). Examining the preconditions for **create**(T) in A_{h_L}, we see that *status*(T) is *requested* because *curT*(p) $= T$ and *status*(p) $=$ *requested*. We know that $\|$*using-resources*$\|$ < TOTAL_RESOURCES because for a node T of A_{h_L} to be in *using-resources* it must have *status running*, and thus there must be a distributed client automaton with *curT* $= T$ and *status running*. Since there are TOTAL_RESOURCES distributed client automata and p has *status requested*, there are at most TOTAL_RESOURCES $-$ 1 nodes of A_{h_L} in *using-resources* and this precondition is satisfied. The last precondition is an underspecified predicate, and we see that it has as its upper bound the previous precondition (that is, the previous precondition must be true if *create-ok* is true). Thus, we choose that it be enabled in the state $S(s')$. Note that this is the only place were we so choose a value for *create-ok*, so no conflicts are possible. Thus, **create**(T) is enabled in A_{h_L}.

The value of *status*(p) changes from *requested* to *running*, and this causes *status*(T) to become *running* at the high level. Thus, $S(s)$ is $S(s')$ with this one change, and this is exactly the postcondition of **create**(T) in A_{h_L}.

commit(T', **results**, p): This maps to **commit**(T', **results**). Since the precondition for this action is met, *status*(T') in A_{h_L} is *committing*, and (T', *results*) $\in$ *saved-results*, by the second point of the state mapping. Thus, **commit**(T', **results**) is enabled in A_{h_L}.

At A_D, the <DONE> is removed from the queue, the recorded status of T' is changed to *committed*, and the *precious* variable in T''s recorded status is set to *nil*. If the *masterp* component of the message is set, *master* is set to *parent*(T) or *void* (the latter if *parent*(T) $\in$ *precious*). The effect of these changes on the high-level state are that *status*(T') is *committed*, (T, *results*) is no longer in *saved-results*, T is no longer in *precious*. If the *masterp* component of the message was set, then that message contains a deadlock master token, and therefore there are no other tokens in transit, and no process has *curT* = *master*. Thus, *master* = T' in $S(s')$ of A_{h_L}. If *preciousp*(p), then *parent*(T') ∈ *precious* in A_{h_L}, and *master* is set to *nil* in A_{h_L}. Examining the distributed client automaton code, we see that *master*(p) is set to *nil*, and thus *master* is *nil* in $S(s)$ of A_{h_L}. Otherwise (*parent*(T') is not precious), *master* is set to *parent*(T') in A_{h_L}. Examining the distributed client automaton code, we see that *master*(p) is set to *parent*(T'), and thus the value of *master* of A_{h_L} becomes *parent*(T') in $S(s)$.

die(T, p): This maps to die(T).

Examining the preconditions for die(T, p), we see that they map straightforwardly to the preconditions of die(T) in A_{h_L}.

At distributed deadlock recovery automaton p, the state is changed by resetting the local state (basically everything but the recorded identity of the deadlock master and message queues), and enqueueing a <DIE-ACK>. This causes *status*(T) at A_{h_L} to become *dead* and *death-status* to become *nil*. Also, it causes *status*(T'), for all children T' of T, to become *not-started*, since that is the default state and *child-status* of p was set to ∅.

receive-do-work(T, p): This maps to the empty sequence. It changes the low-level state by removing the <DO-WORK> message, and setting the local variables appropriately. Node T still has *status requested* at the high level, because this merely replaces the second reason for this with the third in the state mapping (a <DO-WORK> existing turns into a process having the node with *status requested*). Thus, the high-level state is unchanged.

restart(T, p): If *preciousp*(p) is *true*, this maps to restart-precious(T, T'), and restart(T, T') if not. All of the preconditions of the high-level action except the penultimate can straightforwardly seen to be satisfied. If *preciousp*(p) is *nil*, this precondition is satisfied for restart(T). Otherwise, we observe that no entry in *child-status* can have *preciousp true*, because none has *status running*, since *alive-children-count(*child-status*)* is 0. If one did, then that node would be in *precious* in A_{h_L} and have *status* other than *running*. But this cannot be, by Lemma 4.3. Thus, *children(*parent*(T))* ∩ *precious* is the empty set, and the penultimate precondition is also satisfied in this case. Thus, the corresponding action of A_{h_L} is enabled.

Consider the case in which *parent*(T) is not precious. Then the effect of the low-level action on the low-level state is to change the status of the selected child to *requested* from *dead*. By inspecting the code for the high-level action, one can see that the preconditions are satisfied in all image states of states in which the low-level preconditions are satisfied, and the effect of the action at the high-level is to change the status of the child from *dead* to *requested*. If *parent*(T) is precious (that is, if *preciousp*(p) is *true*), since *curT* = *parent*(T), by the preconditions of the action, T is additionally made precious at the low-level, so $S(s)$ is also the state of A_{h_L} obtained by taking the mapped action from $S(s')$.

become-master(T, p): If *master*(p) = *nil*, the processor named in the message at the head of p's input queue is p, *master-attempted*(p) is *true*, and *curT*(p) = T, this maps to become-master(T). Otherwise it maps to the empty sequence.

If the action does not map to the empty sequence, then by Lemma 6.10, *status*(*p*) = *running* and *preciousp*(*p*) = *nil*, and thus the first three preconditions (the truth of the first one follows directly those of the distributed client automaton) of **become-master**(T) of A_{h_L} are satisfied. Because the last two preconditions are underspecified predicates with upper bound *true*, we decide that they are enabled in $S(s')$. As for **create**, this is the only place where we make such a decision, and therefore we have not overspecified these underspecified predicates.

The state of distributed deadlock recovery automaton p is changed so that *master* becomes T. Since $T = curT$, this causes the value of *master* of A_{h_L} to become T. We must, however, make sure that the state mapping is not overconstrained, and verify that there is only one distributed deadlock recovery automaton p with *master* $= T = curT$. By Lemma 6.2, this cannot be.

new-master(T, p): This maps to the empty sequence.

By observing the distributed deadlock recovery automaton code, we see that the value of *master* is not changed if *master* = *curT*. Thus, this action has no effect on the value of *master* at A_{h_L}, and no other effects. Thus, the high-level state is unchanged.

receive-am-blocked(T, T', p): If *curT*(*p*) = T', *death-status*(*p*) = *nil* and *master*(*p*) = T', this maps to **begin-die**(T'), and the empty sequence otherwise. These conditions straightforwardly imply that the preconditions for **begin-die**(T') are true in $S(s')$.

If the conditions above are met, *death-status*(*p*) is set to *dying*. Thus, $S(s)$ also changes from $S(s')$ by having *death-status*(T') set to *dying*. The **begin-die**(T) action of A_{h_L} changes *death-status*(T) to *dying* as well.

receive-die-down(T, T', p): If *parent*(*curT*(*p*)) = T', this maps to **die-down**(T), and otherwise to the empty sequence.

By examining the automaton code, we see that this action maps to the empty sequence exactly when the **if** clause is false and no changes to the state variables occur. In this case it is clear that the high-level state does not change.

When this action maps to **die-down**(T), we must verify that the high-level action is enabled. More precisely, we must show that the following hold, which then imply directly (through the state mapping) that the high-level action (**die-down**(T)) is enabled.

1. *curT*(*p'*) = T'
2. *death-status*(*p'*) = *dying*
3. *status*(*p*) = *running*
4. *death-status*(*p*) = *nil*

Since *curT*(*p*) $\neq$ *nil* (by the low-level conditions on this action mapping to a non-null sequence at the high level), by Lemma 6.9 *status* $\neq$ *nil*. By the preconditions of this action, *status* $\neq$ *requested*. Thus, *status*(*p*) = *running*.

Since this action maps to **die-down**(T) at the high level, we know that *parent*(*curT*(*p*)) = T'. By Claim 6.5, since *curT*(*p*) = T and *status*(*p*) = *running*, *child-status*(T).*status* =

running at *parent-process*(p). Thus, by Lemma 6.3, $curT(p') = T'$ and *death-status*(p') = *dying*.

Since the <DIE-DOWN> is in the queue and *death-status*(p') is *dying*, *death-status*(p) cannot be dying, since the four conditions of Lemma 6.3 are mutually exclusive.

Thus, the high-level action is enabled.

This action changes *death-status*(T) to *dying*, as does the high-level **die-down** action.

receive-die-ack(T', masterp, p): If *masterp* is *true* and *preciousp*(p) is *nil*, this maps to new-master(*parent*(T')). If *masterp* is *true* and *preciousp*(p) is *true*, this maps to new-master(*nil*). Otherwise, it maps to the empty sequence.

Consider first the case of mapping to the empty sequence. The child is marked dead. This does not change the high-level state, as the child is dead there before (due to the message existing) and it is afterwards as well, due to it being so marked at its parent.

If, on the other hand, *masterp* is *true* and this action maps to new-master(*parent*(T')), a <NEW-MASTER> is enqueued. Inspection of the state mapping shows that this does not change the high-level state.

The value of master is set to either *curT* or *nil*. If it is set to *curT*, the value of *master* in A_{h_L} becomes *curT*. Since the (p', p<DIE-ACK, T', masterp>) tuple exists in a queue in the previous state, *child-status*(T').*status* = *running* at p, by Claim 6.5. By Lemma 6.4, $curT(p) = parent(T')$, and this is the effect of the action of A_{h_L}. If set to *nil*, the value of *master* in A_{h_L} becomes *nil*, because the message previously in the input queue of p was the reason it was T' (there could be no other reasons, since then there would be more than one mastership token, and by Lemma 6.2 there cannot be).

□

Lemma 6.14 *S is a possibilities mapping from A_D to A_{h_L}.*

Proof:

By Lemmas 6.11, 6.12 and 6.13, S satisfies the definition of a possibilities mapping from A_D to A_{h_L}. □

Lemma 6.15 *A_D implements A_{h_L}.*

Proof:

By the previous lemma and Proposition 12 from pages 15–16 of [LT88], which states that A implements B if there exists a possibilities mapping from A to B, A_D implements A_{h_L}. □

Thus, we have shown that every finite behavior of A_D is a finite behavior of A_{h_L}.

6.11 A_D solves A_{h_L}

Before showing that A_D solves A_{h_L}, we will first need to define an intermediate notion of progress — *leads to.*

Definition:

A predicate on the state of A_D being true *leads to*, written $\hookrightarrow$, an action of A_{h_L} if for all fair executions of A_D starting with any reachable state in which the predicate holds, an action that maps to an action of A_{h_L} occurs after finitely many actions. We will write both $P \hookrightarrow \pi_{A_{h_L}}$, meaning some action of A_{h_L}, and $P \hookrightarrow$ **die**(T) means the specific action **die**(T).

Additionally, a predicate on the state of A_D being true *leads to* another predicate on the state of A_D if for all fair executions of A_D starting with any reachable state in which the predicate holds, a state is entered in which the second predicate holds after finitely many actions.

The overall proof strategy will be to show that certain predicates lead to an action of A_{h_L} (e.g., the existence of a deadlock recovery automaton with a dead child and no live children leads to the **restart** action), and that other predicates lead to still other predicates (e.g., a message is present in some queue leads to a state in which that message is no longer present in any queue).

Lemma 6.16 *A_D solves A_{h_L}.*

Proof:

We must show that every fair behavior of A_D is a fair behavior of A_{h_L}. First, we observe that for every fair behavior β of A_D, there exists a fair execution γ of A_D. Because we have exhibited a possibilities mapping from A_D to A_{h_L}, we know that there is a corresponding execution δ of A_{h_L} such that beh$(\delta) = \beta$ (given S and A, we can trivially construct it).

Now, we must show that δ is fair. We know that δ is finite, as we have already shown that every execution of A_{h_L} is finite (Lemma 5.7).

Thus, δ is fair if no action is enabled in the last state. We show that from every state of A_D, either some action occurs that maps to an action of A_{h_L}, or that no action of A_{h_L} is enabled.

Claim 6.17 $\exists p|$status(p) = requested $\hookrightarrow$ **create.**

Proof:

Assume p has *status* = *requested* and *curT* = T. Then **create**(T, p) is enabled. By examining the automaton code, it can be seen that this action remains enabled until it occurs. When it occurs, **create**(T) occurs in δ. □

Claim 6.18 *A given message m being present in some queue* leads to *a state in which m is no longer in any queue. Additionally, a state in which either a* `<DO-WORK>` *or* `<DONE>` *message is present in some queue leads to an action of A_{h_L}.*

Proof:

Any message in an output queue will eventually be sent to the network automaton, since the only precondition of the **send** action is that a message is in the output queue. Any message in the network will eventually be delivered to an input queue. Thus, a state will eventually occur in which some message currently in a queue is in an input queue.

Consider some input queue that is nonempty (call it the input queue of deadlock recovery automaton p). First, we note that for each type of message, only one action can remove it from the input queue. Further, we note that the sole precondition of each of these actions is the existence of the message at the head of the input queue, with the exception of **receive-die-down**. The other precondition of this action is that *status* $\neq$ *requested*, and we have already shown that this holds. Thus, once we show that an action removing a message from an input queue is enabled, we know that it remains enabled until it occurs. We consider the possible values of the message:

<**DO-WORK**, T, **preciousp**>: The **receive-do-work** action is enabled. After this action occurs, *status*(p) will be *requested*, and we have already shown that this cannot be.

<**DONE**, T, **results**, **masterp**>: The **commit**(T', **results**, p) action is enabled. When it occurs **commit** occurs in δ. But we assumed earlier that no more actions occur in δ.

<**WANT-MASTERSHIP**, T, p'>: The **become-master**(T, p) is enabled. When it occurs an action may occur in δ, or the message may merely be removed from the queue. Either another message exists in a queue, or no message does. In the former case, examine that message, and in the latter the claim is proved.

<**NEW-MASTER**, T, p'>: The **new-master**(T, p) action is enabled. When it occurs, the message is removed from the queue.

<**AM-BLOCKED**, T, T'>: The **receive-am-blocked**(T, T', p) action is enabled. When it occurs, the message is removed from the queue.

<**DIE-DOWN**, T>: The **receive-die-down**(T, T', p) action is enabled. When it occurs, the message is removed from the queue.

<**DIE-ACK**, T, **masterp**>: The **receive-die-ack**(T', **masterp**, p) action is enabled. When it occurs, the message is removed from the queue.

Observing that the case analysis for <DO-WORK> and <DONE> showed that an action of A_{h_L} must occur, the second statement of the claim is easily seen to be true.

□

Claim 6.19 work-requested $\neq$ nil $\hookrightarrow$ work-requested = nil.

Proof:

Examining the automaton code, we observe that only the **request-space** action ever sets *work-requested* to any non-*nil* value, and further that each of the following actions set *work-requested* to *nil*: **die**(T, p), **request-commit**(T, **results**, p), **can-do-work**(p, p''), or **no-space**(p).

Thus, **request-space** must occur in β (the execution of A_D under consideration), and further must not be followed by any of the other actions mentioned above.

Recalling the definition of a space allocating automaton given on page 84, we see that in any fair schedule of the automaton every **request-space**(p) not followed by either **die**(T, p) or **request-commit**(T, **results**, p) is followed by either **can-do-work**(p, p') or **no-space**(p).

Thus, one of these actions must occur, and the state following whichever one occurs has *work-requested* = *nil*. □

Claim 6.20 death-status(p) $\neq$ nil $\hookrightarrow \pi_{A_{h_L}}$.

Proof:

First, we observe that if *death-status*(p) is non-*nil*, it must remain so until either **die**, **request-commit**, or **receive-do-work** occurs at p. The former case, **die**, induces a **die** action in A_{h_L}, and the next enqueues a <DONE>, and we have already shown that this leads to an action of A_{h_L}. The last case sets *status* to *requested*, and we have already shown that this leads to the **create** of A_{h_L}. Thus, we will assume that *death-status*(p) remains non-*nil* once non-*nil* for the duration of the proof of this claim (if not, an action of A_{h_L} has occurred and the claim holds).

Examine the value of *child-status*(p). There are two cases:

$\exists T'|$*child-status*(T').*status* = *running* By Lemma 6.3, either a <DIE-DOWN> is en route to the child, the child has *death-status* = *dying*, or either a <DIE-ACK> or <DONE> is returning. In the case that a <DIE-DOWN> message is en route to the child, we can see that when it is removed from the input queue a **die-down** action will occur in A_{h_L}. In the second case, we shift the focus to the child and begin the case analysis again. This process must terminate, as we know that the tree is no more than TOTAL_RESOURCES deep. In the case of a returning <DIE-ACK> message, we wait for it to be received, and again begin the case analysis. Since a client automaton can request only finitely many children, we know that this process must also terminate. In the case of a <DONE> message, we have already shown that a **commit** action will occur.

$\nexists T'|$*child-status*(T').*status* = *running* If *work-requested* $\neq$ *nil*, wait until it becomes *nil* (this will occur, by Claim 6.19). Now, the **die** action for this automaton is enabled, and thus must happen, and thus an action of A_{h_L} occurs.

□

Claim 6.21 *All deadlock recovery automata with* curT *non-*nil *have an outstanding requested child (that is, a non-*nil*value of* child-status*).*

Proof:

Consider one that does not. Then the behavior of the associated client is not fair, because the liveness conditions on client automata are not met. □

Claim 6.22 *A state in which a deadlock recovery automaton p has a non-*nil *value of* child-status leads to *a state in which some element of* child-status *has* status *either* requested *or* running, *or else some action of* A_{h_L} *happens.*

Proof:

If $\exists T \in$ *child-status* : *child-status*(T).*status* $\in$ {*requested, running*}, the condition already holds. If not, every member has *status dead*, and in particular there exists some T such that *child-status*(T).*status*= *dead.* Since *child-status*(T) is non-*nil*, *status*(p) = *running* and *curT*(p) = *parent*(T), by Lemma 6.4. Also, we have already showed that *death-status*(p) must be *nil* (or else an action of A_{h_L} will occur). Thus, the **restart**(T, p) action is enabled, and will remain enabled until some element of *child-status* has *status* other than *dead.* □

Claim 6.23 *A processor p exists that is unoccupied and allocated leads to an action of* A_{h_L}.

Proof:

Since p is allocated and unoccupied, the schedule of the space allocation automaton must contain a **can-do-work**(p', p) not followed by a **die** or **request-commit** action occurring at either p' or p. Thus, either a `<DO-WORK>` is en route to p, or has arrived. In either case, we have already shown that the **create** action must occur at p. □

Claim 6.24 *A deadlock recovery automaton p exists such that* child-status(T).status = requested leads to *an action of* A_{h_L} *or the* **request-space**(p) *action of* A_D.

Proof:

Observing the automaton code, we see that only the **can-do-work, die** and **commit** actions can change the status of a child away from requested (**restart** is not enabled because a child has *status requested.* The latter two actions induce an action of A_{h_L}.

Consider the value of *work-requested*(p). If it is not *nil*, wait until it becomes *nil*, since we have already shown that this must happen. If *nil*, the **request-space**(p) action is enabled (unless *death-status*(p) is non-*nil*, in which case an action of A_{h_L} must occur), and must remain so unless either *death-status* becomes non-*nil* or an action of A_{h_L} is induced. Thus, either **request-space**(p) occurs or an action if A_{h_L} is induced. □

Claim 6.25 *A deadlock recovery automaton p exists such that* child-status(T).status = requested *and there exists an unallocated process* p' leads to *an action of* A_{h_L}.

Proof:

First, observe that by the definition of *allocated*, p' will remain unallocated until either a **can-do-work**(p'', p') action occurs or **create**(T', p') occurs. After the former occurs, p' is unoccupied and allocated, and by the previous claim **create** will occur. Thus, for the remainder of the proof of this claim, we can assume p' remains unallocated (since if it does not, the consequent holds). We further assume that **can-do-work** does not occur, since if it does a `<DO-WORK>` message is enqueued, causing a **create**.

By the previous claim, a **request-space**(p) action must occur. Then, the space allocation automaton must take a **can-do-work** action, by the definition of a space allocation automaton, since p' is still unallocated. This in turn will induce a **create** action, since a `<DO-WORK>` message will be enqueued. □

Claim 6.26 *A state in which all nodes are allocated* leads to *either an action of* A_{h_L} *or a state in which all nodes are occupied.*

Proof:

Consider a node p which is unoccupied and allocated. By the definitions of occupied and allocated and space allocating automata, p was unoccupied when the **can-do-work** action making it allocated occurred. At this point a `<DO-WORK>` was enqueued, and we have already showed that the existence of this message in a queue leads to the **create** action.

If no such node can be found, then all nodes must be occupied. □

Claim 6.27 *A state in which all nodes are occupied and some node has an element of* child-status *with* status requested leads to *an action of* A_{h_L}.

Proof:

If a state is entered in which some node is no longer occupied, then the **request-commit** or **die** action must have occurred at that node, by the definition of occupied. In the former case, a `<DONE>` action is enqueued, which will induce a **commit** action, and in the latter an action of A_{h_L} has occurred. Thus, we may assume that all nodes remain occupied.

Choose a deadlock recovery automaton p such that *child-status*$(p)(T)$.*status* = *requested*, $\not\exists T'|$*child-status*$(p)(T')$.*status* = *running* and *preciousp*(p) = *nil*. To see that such an automaton exists, first observe that there are at most TOTAL_RESOURCES − 1 automata with *preciousp* = *true*, since if there were TOTAL_RESOURCES the lowest-level one could not have *child-status* non-*nil* (by the definition of client automata, the definitions of S and A, the high-level safety properties already shown about preciousness, and the observation that only the **request-create-child** action adds elements to the initially empty *child-status* set). Now, consider any non-precious automaton with an element of *child-status* with *status running*. If none exists there must be at most one non-precious automaton with an element of *child-status* with *status requested* (or else the behavior of the associated client is not fair, by Claim 6.21). If one exists, either a `<DO-WORK>` is en route to some automaton or some automaton has *curT* the value of the running child of the parent under consideration. The former case cannot occur, however, since it would imply that an automaton is unoccupied, and by hypothesis this is not so. Now, shift the focus to the running child that has just been found. In this way, we will eventually come to an automaton with no elements of *child-status* with *status running*, and some element with *status requested*. In addition, this node will not be precious.

Since some node p has an element of *child-status* with *status requested*, either an action of A_{h_L} occurs or the **request-work**(p) action occurs, by Claim 6.24.

The space allocation automaton must perform the **no-space**(p) action, since it may not perform a **can-do-work** action, and must perform one of the two actions. Now, we do a case analysis:

master-process(p) = *nil* Examining the automaton code for the **no-space** action, we see that either a `<WANT-MASTERSHIP>` message is enqueued, or *master-attempted(p)* is non-*nil*. Examining the automaton code, we see that *master-attempted(p)* is only set to a non-*nil* value when a *WANT-MASTERSHIP* message is sent, and is cleared by exactly the following actions: **request-commit** (which leads to an action of A_{h_L}), **die** (which directly induces an action of A_{h_L}), and **become-master**(p).

Thus, if *master-attempted(p)* is non-*nil*, there is a `<WANT-MASTERSHIP, T, p>` message sent by p that has not yet been received by p. If this message is received by

p before any other <WANT-MASTERSHIP> messages, p will become deadlock master, since in that case *master-attempted* will still be non-*nil*, and the $curT = T$ condition must still be satisfied (or else a **request-commit** must have occurred). Otherwise, some other node will become deadlock master on receiving its own <WANT-MASTERSHIP> message. If for some reason the receipt of the first such message causes the originating node to send a <NEW-MASTER, *nil*, *nil*> message instead, we can then begin the analysis from the beginning, except that this time no <NEW-MASTER, *nil*, *nil*> messages can be sent, since they require that a **request-commit** action occur at p between the sending and receiving of the <NEW-MASTER, T, p> at p.

Since the <WANT-MASTERSHIP> message that was first in the serialized ordering of messages was broadcast, it will eventually be received at all processes. Wait until this occurs. At this point, all processes have the correct notion of the identity of the deadlock master, because no **request-commit** or **die** actions have occurred (or else an action of A_{h_L} would be induced).

$masterp(p) = p'$, and $masterp(p') \neq p'$ Wait until all <WANT-MASTERSHIP> messages extant, if any, are delivered. If this case no longer applies, restart the case analysis. Now, process p has recorded that the deadlock master is p', but p' does not have recorded that it is master. Since p' must also have received the <WANT-MASTERSHIP> message containing p' as a processor identifier, but is no master, it must have either enqueued a <NEW-MASTER, *nil*, *nil*>, or a **request-commit** or **die** action must have occurred. Either action occurring would lead to an action of A_{h_L} occurring, so we must assume the <NEW-MASTER, *nil*, *nil*> message was sent. Wait until this message is received at all processors, and restart the case analysis. We know that this situation cannot recur because no automaton may perform the **request-commit** or **die** actions, and once a process sends a <WANT-MASTERSHIP> message, it will never send <NEW-MASTER> unless the client automaton running at it requests to commit or dies (since otherwise the value of $curT$ will be unchanging).

$masterp(p) = p'$, and $masterp(p') = p'$ Process p sends a <AM-BLOCKED> message to the process denoted by $masterp(p)$. This process must still think itself to be deadlock master, and still have the same associated client automaton (since otherwise an **request-commit** or **die** action must have occurred). On receipt, the process sets *death-status*(*masterp*) to *dying*, and we have already shown that this induces an action of A_{h_L}.

□

Claim 6.28 *Any state leads to an action of A_{h_L} unless the behavior of some client automaton is not fair.*

Proof:

Assume that the behavior of every client automaton is fair. Then, by Claims 6.21 and 6.22, some automaton has a value of *child-status* with *status requested.* Now, either all nodes are occupied, or all nodes are not occupied. By Claim 6.25 and the previous claim, some action of A_{h_L} must occur. □

Thus, every fair behavior of A_D is a fair behavior of A_{h_L}. □

6.12 A_D preserves client well-formedness

Lemma 6.29 *Every execution of A_D is well-formed.*

Proof:

The proof is by induction on the length of executions.

Basis:

Examine the shortest execution, consisting of only the start state, and observe that every block of every client automaton is empty, and that it is therefore well-formed.

Inductive Step:

Given an execution α of A_D that is well-formed, we show that $\alpha\pi$ is well-formed if $\alpha\pi$ is an execution of A_D, by enumeration of possible values of π.

Because of the definition of well-formedness (page 81) for distributed client automata, we need only consider values of π which are external actions of client automata. Further, because distributed client automata preserve well-formedness, we need not consider output actions of client automata.

Because we have given a mapping of executions of A_D to executions of A_{h_L}, and all executions are A_{h_L} are well-formed, we need only show that the additional constraint that the value of p is the same for all actions within a block of a client automaton is satisfied.

create(T, p): This action begins a block, and thus the new execution is well-formed if the old one is.

commit(T', results, p): Since the action occurred, there was a `<DONE, T', results, masterp>` message in the input queue of p. By Claim 6.5, *child-status*$(p)(T')$.*status* = *running*. Thus, *curT*(p) is *parent*(T') and *status*(p) is *running*, by Lemma 6.4.

Thus, the create(*parent*(T'), p) action must have occurred more recently than any die(*parent*(T'), p) or request-commit(*parent*(T'), results, p), or else *curT*(p) would be *nil*. Thus, the current block of client *parent*(T') already has value p associated with it.

die(T, p): By the definition of well-formedness of client automata, a die action does not belong to any block. Thus, adding this action to a well-formed execution produces a well-formed execution.

□

6.13 A_D is correct

Theorem 6.30 *A_D is correct.*

Proof:

Since A_D solves A_{h_L} and A_D preserves client well-formedness, the distributed version of the deadlock recovery algorithm, A_d, is correct, by Lemma 6.1. □

Chapter 7
Conclusion

This chapter contains a summary of the thesis and suggests topics for future research.

7.1 Summary

We started with a problem that arose naturally during the process of designing a system for highly-reliable high-throughput computing. Then, we gave formal correctness conditions for the problem in terms of I/O Automata. This required the introduction of client automata, which model the behavior of the user's program for which the deadlock recovery algorithm functions as a scheduler. We placed conditions, both safety and liveness, on client automata in order to be able to show that the combined system was correct. The use of I/O Automata helped greatly in being sure that the conditions were clearly expressed, and allowed the intuition of what was "reasonable" for a client automaton to do to be expressed both formally and reasonably easily in a way that can be precisely understood by the reader.

After presenting a high-level algorithm to solve the problem, we proved safety properties about the algorithm. Then, we showed liveness properties of the algorithm. In the liveness proof, we used a modified version of the standard variant function technique; instead of presenting a metric on states with no infinite decreasing chains, we show that no execution of the algorithm contains an infinite decreasing chain. Also, we show that every locally-controlled action of the algorithm decreases the metric, rather than showing that some action decreasing the metric occurs infinitely often.

Then, a distributed version of the algorithm and a possibilities mapping from the distributed version to the high-level version was presented. Finally, we showed liveness properties of the distributed version, completing the proof that the distributed version of the algorithm is correct.

7.2 Future work

This section suggests some areas for future work.

7.2.1 Proofs of time bounds

We showed that an algorithm had certain liveness properties, meaning that there were no infinite executions that did not solve the problem. We did not attempt to give time bounds on how long the algorithm took to solve the problem; this seems to be a much harder problem.

Also, in the current proof we did not place a bound on the number of requests each client automaton may make; Proving time bounds without such a limit will require defining just how the running time of an algorithm that must respond to an unbounded but finite number of requests should be measured.

7.2.2 Issues in liveness proofs

We chose the modified variant function strategy for the liveness proof; this seemed to place the fewest constraints on the client automata. For example, a nondeterministic client automaton that requests one child the first time it is started, two the second time, and so on, is acceptable under the current set of definitions. One might instead require that the algorithm satisfy requests within a finite time of the cessation of requests by each client, or some other conditions.

One could further explore the variant technique; one idea that was considered in this thesis was to have each request made by a client automaton increment the variant function. Then, one could show that the algorithm would terminate as long as the number of requests made by client automata were finite.

7.2.3 Automation of proofs

Many of the arguments in the safety proofs are trivial, but one must write them all out to be sure that the proof is really a proof. One could investigate using automated theorem proving techniques to verify invariants, having the automated prover attempt to show the inductive step after the human doing the proof provided a set of invariants (such as those in Lemma 4.3). The prover could probably be able to show some of the components of the large invariant, and not others. The human could then examine each place where the proof did not go through, and determine (as I often did when doing the proof in Chapter 4) that another invariant is needed. Alternatively, it could be determined that that component can indeed be shown from the others already there, but that the theorem prover could not do it — these are often the interesting cases anyway — and that part of the proof could be done by hand.

The same techniques could be applied to the problem of showing that a given mapping is a possibilities mapping.

Bibliography

[Awe85] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM*, 32(4):804–823, October 1985.

[Eve79] Shimon Even. *Graph Algorithms.* Computer Science Press, 1979.

[Har85] Richard E. Harper. *The FTPP Cluster.* Working paper, Charles Stark Draper Laboratory, April 1985.

[Har87] Richard E. Harper. *Critical Issues in Ultra-Reliable Parallel Processing.* PhD thesis, Massachusetts Institute of Technology, June 1987.

[Kim86] Sam Sang Kyu Kim. *A Computational Study of a Parallel Simulated Annealing Algorithm for the Traveling Salesman Problem.* Master's thesis, Massachusetts Institute of Technology, June 1986.

[LG85] Kwei-Jay Lin and John D. Gannon. Atomic remote procedure call. *IEEE Transactions on Software Engineering*, SE-11(10):1126–1135, October 1985.

[LMWF88] Nancy A. Lynch, Michael Merritt, William Weihl, and Alan Fekete. *A Theory of Atomic Transactions.* Technical Report MIT/LCS/TM-362, Massachusetts Institute of Technology, June 1988.

[LT87] Nancy A. Lynch and Mark R. Tuttle. *Hierarchical Correctness Proofs for Distributed Algorithms.* Technical Report MIT/LCS/TR-387, Massachusetts Institute of Technology, April 1987.

[LT88] Nancy A. Lynch and Mark R. Tuttle. *An Introduction to Input/Output Automata.* Technical Report MIT/LCS/TM-373, Massachusetts Institute of Technology, November 1988. TM-351 Revised.

[Seg83] Adrian Segall. Distributed network protocols. *IEEE Trans. Info. Theory*, IT-29(1):23–35, January 1983. Some details in technical report of same name, MIT Lab. for Info. and Decision Syst., LIDS-P-1015; Technion Dept. EE, Publ. 414, July 1981.

[Tro87] Gregory D. Troxel. Detection of and recovery from deadlock algorithm for deadlock recovery in a system using remote procedure calls. S.B. thesis, Massachusetts Institute of Technology, June 1987.

[Wel88] Jennifer L. Welch. *Topics in Distributed Computing: The Impact of Partial Synchrony, and Modular Decomposition of Algorithms.* PhD thesis, Massachusetts Institute of Technology, June 1988.

[WLL88] Jennifer Lundelius Welch, Leslie Lamport, and Nancy Lynch. A lattice-structured proof of a minimum spanning tree algorithm. 7th PODC, January 1988.

OFFICIAL DISTRIBUTION LIST

DIRECTOR Information Processing Techniques Office Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209	2 Copies
OFFICE OF NAVAL RESEARCH 800 North Quincy Street Arlington, VA 22217 Attn: Dr. Gary Koop, Code 433	2 Copies
DIRECTOR, CODE 2627 Naval Research Laboratory Washington, DC 20375	6 Copies
DEFENSE TECHNICAL INFORMATION CENTER Cameron Station Alexandria, VA 22314	12 Copies
NATIONAL SCIENCE FOUNDATION Office of Computing Activities 1800 G. Street, N.W. Washington, DC 20550 Attn: Program Director	2 Copies
HEAD, CODE 38 Research Department Naval Weapons Center China Lake, CA 93555	1 Copy