RWTH Aachen

Department of Computer Science
Technical Report

Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

Neeraj Mittal, S. Venkatesan, Felix Freiling and Lucia Draque Penso

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2005-12

RWTH Aachen · Department of Computer Science · May 2005

Efficient Reductions for Wait-Free Termination Detection in Faulty Distributed Systems

Neeraj Mittal[2⋆], Felix Freiling[1], S. Venkatesan[2], and Lucia Draque Penso[1⋆⋆]

[1] Computer Science Department, RWTH Aachen University,
D-52056 Aachen, Germany
[2] Department of Computer Science, The University of Texas at Dallas,
Richardson, TX 75083, USA

Abstract. We investigate the problem of detecting termination of a distributed computation in asynchronous systems where processes can fail by crashing. More specifically, for both fully and arbitrarily connected communication topologies, we describe efficient ways to transform any *fault-sensitive* termination detection algorithm $\mathcal{A}$, *that has been designed for a failure-free environment*, into a *wait-free* termination detection algorithm $\mathcal{B}$, *that tolerates up to any number of process crashes*. The transformations are such that a competitive fault-sensitive termination detection algorithm $\mathcal{A}$ results in a competitive wait-free termination detection algorithm $\mathcal{B}$. Furthermore, they work whether the termination detection is *effective* (allowing messages in-transit from crashed processes towards a live one able to ignore them) or *strict* (not allowing messages in-transit towards a live process). Finally, though we focus on crash failures, we also discuss how to tolerate message omissions, and how they impact on the performance.
Let $\mu(n, c, M)$ and $\delta(n, c, M)$ denote message complexity and detection latency of $\mathcal{A}$ when the system has n processes and c bidirectional reliable channels, and when the distributed computation exchanges M application messages. For fully connected communication topologies, the message complexity and the detection latency of $\mathcal{B}$ are at most $O(n + \mu(n, c, 0))$ messages per fault more and $O(\delta(n, c, 0))$ time units per fault more than those of $\mathcal{A}$, while for arbitrary ones, they are at most $O(n \log n + c + \mu(n, c, 0))$ messages per fault more and $O(n + \delta(n, c, 0))$ time units per fault more than those of $\mathcal{A}$. Moreover, for both cases, the overhead (that is, the amount of control data piggybacked) increases by only $O(\log n)$ bits per fault on an application message and at most $O(\log n)$ bits per fault plus $O(\log M)$ on a control message.
We also prove that in a crash-prone distributed system, irrespective of the number of faulty processes, the *perfect failure detector* is the weakest failure detector for solving the effective-termination detection problem, whereas a *failure detection sequencer* is both necessary and sufficient for solving the strict-termination detection problem. This guarantees that our transformation method, which requires a perfect failure detector, does not demand stronger than necessary assumptions.

Key words: asynchronous system, failure detector, fault-tolerance, faulty processes, reduction, termination detection, wait-free algorithm

1 Introduction

The problem of detecting termination of an underlying distributed application still remains as one of the main problems in distributed systems, despite having been independently proposed more than two decades ago by Francez [11] and Dijkstra and Scholten [9]. As expected, the problem has been extensively studied since then, and a variety of efficient protocols have been designed for termination detection (*e.g.*, [11, 9, 25, 22, 27, 20, 8, 21, 31, **?**,15, 5, 33, 29, 18, 30, 28, **?**,19, 32, 24, 23]). Interestingly, most of the termination detection algorithms in the literature have been developed assuming that both processes and channels stay operational throughout an execution and not much effort has

⋆ Neeraj Mittal was supported by Deutsche Forschungsgemeinschaft (DFG) when visiting RWTH Aachen University.
⋆⋆ Lucia Draque Penso was supported by Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg "Software for Mobile Communication Systems" at RWTH Aachen University.

been done towards obtaining efficient fault-tolerant termination detection algorithms. Real-world systems, however, are often prone to failures. For example, processes may fail by crashing and channels may be lossy.

In this paper, we investigate the termination detection problem when any number of processes can fail by crashing and the (fault-tolerant) underlying distributed application is not restarted as a result. Hence, we search for solutions which are *wait-free* [14]: any live process finishes in a fixed number of steps regardless of delays or failures by other processes, or equivalently, in a crash-prone distributed system, regardless of the number of process crashes. In particular, we do that by efficiently reducing the wait-free termination detection problem in a crash-prone distributed system to the *fault-sensitive* (that is, fault-*in*tolerant) case, making it possible to have a competitive wait-free termination detection algorithm $\mathcal{B}$ out of a competitive fault-sensitive termination detection algorithm $\mathcal{A}$. More precisely, for both fully and arbitrary connected topologies, we show how to efficiently transform any fault-sensitive termination detection algorithm $\mathcal{A}$, that has been designed for a failure-free environment, into a wait-free termination detection algorithm $\mathcal{B}$, that tolerates up to any number of process crashes. We also discuss how both reductions may be extended to tolerate message omissions and how their performance gets impacted.

Given a crash-prone distributed system, let n be the total number of processes, c be the number of bidirectional reliable channels, f be the actual number of processes that fail during an execution, M be the number of application messages exchanged by the underlying distributed application, $\mu(n, c, M)$ and $\delta(n, c, M)$ be the message complexity and the detection latency of $\mathcal{A}$, and $\alpha(n, c, M)$ and $\beta(n, c, M)$ be the amount of control data piggybacked, called overhead, on an application message and on a control message of $\mathcal{A}$.

Our reductions perform as follows. Note that, for most competitive termination detection failure-free algorithms, $\mu(n, c, 0)$ varies from $O(n)$ to $O(c)$ and $\delta(n, c, 0)$ is $O(1)$, $1 \leq c \leq n(n-1)/2$. The message complexity and the detection latency of $\mathcal{B}$ are at most $O(n + \mu(n, c, 0))$ messages per fault more and $O(\delta(n, c, 0))$ time units per fault more than those of $\mathcal{A}$ for fully connected communication topologies, and at most $O(n \log n + c + \mu(n, c, 0))$ messages per fault more and $O(n + \delta(n, c, 0))$ time units per fault more than those of $\mathcal{A}$ for arbitrarily connected ones. Furthermore, the overhead increases by only $O(\log n)$ bits per fault on an application message and either $O(\log n)$ or $O(\log M)$ bits per fault on a control message. In particular, the overhead of $O(\log M)$ applies only to those control messages that are exchanged whenever a crash is detected. Clearly, our reductions do not impose any additional overhead on the system (besides that imposed by $\mathcal{A}$) if no process actually crashes during an execution.

One of the earliest fault-tolerant termination detection algorithms was proposed by Venkatesan [31], which was derived from the fault-sensitive termination detection algorithm by Chandrasekaran and Venkatesan [5]. Venkatesan's algorithm achieves crash-tolerance by replicating state information at multiple processes that communicate through bidirectional reliable FIFO channels in a k-connected communication topology, $1 \leq k \leq n$. However, to become wait-free in a crash-prone system, it requires not only a fully connected communication topology, but also an atomic send of n recipient-distinct messages. Its wait-free version has a message complexity of $O(nM + n^2)$ and a detection latency of $O(M)$.

Lai and Wu [18] and Tseng [30] modify fault-sensitive termination detection algorithms by Dijkstra and Scholten [9] and Huang [15], respectively, to derive two different wait-free termination detection algorithms for crash-prone systems. Both algorithms assume that the communication topology is fully connected. However, unlike Venkatesan's algorithm, both have a low message complexity of $O(M + fn + n)$ and detection latencies of $O(n)$ and $O(f)$, respectively. Despite the lower detection latency, note that the algorithm by Tseng has higher application and control

Algorithm	Message Complexity	Detection Latency	Message Overhead		Assumptions
			Application	Control	
Venkatesan* [31]	$O(nM + n^2)$	$O(M)$	n-way duplication	$O(\log n + \log M)$	Fully connected topology + Atomic n-send† + FIFO channels
Lai and Wu [18]	$O(M + fn + n)$	$O(n)$	0	$O(f \log n + \log M)$	Fully connected topology
Tseng [30]	$O(M + fn + n)$	$O(f)$	$O(M)$	$O(f \log n + nM)$	Fully connected topology
Our Approach for Fully [this paper] with Dijkstra and Scholten [9]	$O(M + fn + n)$	$O(n)$	$O(M \log n)$	$O(f \log n + \log M)$	Fully connected topology
Our Approach for Fully [this paper] with Huang [15]	$O(M + fn + n)$	$O(f)$	$O(M \log n)$	$O(f \log n + \log M)$	Fully connected topology
Our Approach for Fully [this paper]	$O(\mu(n, c, M))$ + $O(f(n + \mu(n, c, 0)))$	$O(\delta(n, c, M))$ + $O(f\delta(n, c, 0))$	$O(\alpha(n, c, M) + f \log n)$	$O(\beta(n, c, M) + f \log n)^1$ and $O(f \log n + \log M)^2$	Fully connected topology
Our Approach for Arbitrarily [this paper]	$O(\mu(n, c, M))$ + $O(f(c + \mu(n, c, 0)))$ + $O(fn \log n)$	$O(\delta(n, c, M))$ + $O(f(n + \delta(n, c, 0)))$	$O(\alpha(n, c, M) + f \log n)$	$O(\beta(n, c, M) + f \log n)^1$ and $O(f \log n)^2$	Arbitrarily connected topology + FIFO channels
Our Approach for Arbitrarily [this paper] with Neeraj and Venkatesan [23]	$O(M)$ + $O(f(c + n \log n))$	$O(fn)$	$O((M + f) \log n)$	$O(c + f \log n)$	Arbitrarily connected topology + FIFO channels
Shah and Toueg [26]	$O(cM)$	$O(n^2)$	$O((M + f) \log n)$	$O(c + n \log n)$	Arbitrarily connected topology + FIFO channels
Gärtner and Pleisch [12]	$O((n + c)M)$	$O(n^2)$	$O((M + f) \log n)$	$O(c + n \log n)$	Arbitrarily connected topology + Failure detection sequencer‡

*Venkatesan's algorithm has non-trivial preparation cost of $\approx 3nM + n(n-1)$

[1] overhead for control messages of the fault-sensitive termination detection algorithm

[2] overhead for control messages exchanged whenever a crash is detected

† atomic send of n recipient-distinct messages

‡ failure detector sequencers are able to emulate FIFO channels

n: initial number of processes in the system
c: number of channels in the connected communication topology, equal to $n(n-1)/2$ for fully ones
f: actual number of processes that crash during an execution
M: number of application messages exchanged
$\mu(n, c, M)$: message complexity of the fault-sensitive termination detection algorithm
$\delta(n, c, M)$: detection latency of the fault-sensitive termination detection algorithm
$\alpha(n, c, M)$: application message overhead of the fault-sensitive termination detection algorithm
$\beta(n, c, M)$: control message overhead of the fault-sensitive termination detection algorithm

Table 1. Comparing wait-free termination detection algorithms for crash-prone distributed systems.

message overheads of $O(M)$ and $O(f \log n + nM)$, in comparison to 0 and $O(f \log n + \log M)$ by Lai and Wu.

Shah and Toueg give a crash-tolerant algorithm for taking a consistent snapshot of a distributed system in [26]. Their algorithm is derived from the fault-sensitive consistent snapshot algorithm by Chandy and Lamport [6]. As a result, each invocation of their consistent snapshot algorithm may generate up to $O(c)$ control messages. It is easy to verify that, when their algorithm is used for termination detection, the message complexity of the resulting algorithm reaches up to $O(cM)$ in the worst-case.

Similarly, Gärtner and Pleisch [12] give an algorithm for detecting an arbitrary stable predicate in a crash-prone distributed system. (Note that termination is a stable property.) In their algorithm, every relevant local event is *reliably and causally broadcast* to a set of monitors, thereby increasing the message complexity significantly.

Typically, generalized reductions tend to be inefficient compared to customized ones. However, when our transformation for a fully connected communication topology is applied to fault-sensitive termination detection algorithms by Dijkstra and Scholten [9] and Huang [15], the resulting wait-free algorithms for crash-prone systems are comparable to those by Lai and Wu [18] and Tseng [30] and even outperform the one by Venkatesan [31], in terms of message complexity and detection latency. Moreover, when our transformation for an arbitrarily connected communication topology is applied to the fault-sensitive termination detection algorithm by Neeraj and Venkatesan [23], the resulting wait-free algorithm for crash-prone systems outperforms those by Shah and Toueg [26] and Gärtner and Pleisch [12].

More specifically, when our transformation for a fully connected communication topology is applied to Dijkstra and Scholten's algorithm [9], the resulting algorithm has the same message complexity, detection latency and control message overhead as the algorithm by Lai and Wu [18]. However, the application message overhead is higher for our algorithm. Likewise, when our transformation for a fully connected communication topology is applied to Huang's algorithm [15], the resulting algorithm has the same message complexity and detection latency as that of the algorithm by Tseng [30]. Our algorithm has slightly higher application message overhead but much lower control message overhead. Higher application message overheads are not surprising because our transformation is general; it works for any termination detection algorithm. Finally, when our transformation for an arbitrarily connected communication topology is applied to the algorithm by Neeraj and Venkatesan [23], the resulting algorithm has better message complexity, detection latency and (application and control) message overhead than those of the algorithms by Shah and Toueg [26] and Gärtner and Pleisch [12]. For comparison between various wait-free termination detection algorithms for crash-prone distributed systems, please refer to Table 1.

The main idea behind our approach is to *restart* the fault-sensitive termination detection algorithm *whenever a new failure is detected* . A separate mechanism is used to account for those application messages that are in-transit when the termination detection algorithm is restarted. Interestingly, it works whether the termination detection is *effective* (allowing messages in-transit from crashed processes towards a live one able to ignore them) or *strict* (not allowing messages in-transit towards a live process).

We build upon the work by Wu *et al* [33]. We do this in the context of the failure detector hierarchy proposed by Chandra and Toueg [4], a way to compare problems based on the level of synchrony required for solving them. More precisely, we show that in a crash-prone distributed system, irrespective of the number of faulty processes, a *perfect failure detector* [4] is the weakest synchrony assumption (therefore, needed and sufficient) for the effective-termination detection problem to be solvable, whereas a *failure detection sequencer* [12] is the weakest one for the strict-termination detection problem to be solvable. Despite not being itself a failure detector, note that it is straightforward to verify that a failure detection sequencer has a higher level of synchrony than a perfect failure detector, as the first is capable of emulating the second. This result can be used to further understand the relationship between (effective or strict) termination detection and other problems in fault-tolerant distributed computing, such as consensus. Moreover, it proves that our approach, which makes use of a perfect failure detector, does not require stronger than necessary assumptions.

Arora and Gouda [1] also provide a mechanism to reset a distributed system. However, their work is completely distinct from ours in many ways.

First of all, the semantics of their reset operation is totally different from the semantics of our restart operation. More specifically, if their reset mechanism is applied to our system, then it will not only reset the termination detection algorithm but will also reset the underlying distributed application, whose termination is to be detected. Furthermore, application messages exchanged by the underlying distributed application before it is reset will be discarded. Thus, if a failure occurs near the completion of the underlying distributed application, the entire work needs to be redone if the distributed reset procedure is used. In contrast, in our case, the underlying distributed application continues to execute without interruption. Therefore, in our case, application messages exchanged before the termination detection algorithm is restarted, especially those exchanged between correct processes, cannot be ignored. In short, Arora and Gouda's approach is more suitable for underlying distributed applications that can be reset on occurrence of a failure whereas our approach is more suitable for those that continue to execute despite failures.

Second, in their approach, the system may be reset more than once due to the same failure. This may happen, for example, when multiple processes detect the same failure at different times. Third, their reset operation, which is self-stabilizing in nature, is designed to tolerate much broader and more severe kinds of faults, such as restarts and arbitrary state perturbations. Not surprisingly, their reset operation has higher message and time complexities than our restart operation.

Finally, their approach is non-masking fault-tolerant, which implies that the safety specification of the application may be violated temporarily, even if there is a single crash fault. When translated to our problem, this means that the termination detection algorithm may wrongly announce termination, a case which our approach avoids.

This paper is organized as follows. In Section 2, we present our crash-prone distributed system model and describe what it means to detect (effective or strict) termination in such a system. In Section 3, we discuss our reductions for both fully and arbitrarily connected communication topologies, and comment on how they may be extended to tolerate message omissions. In Section 4, we determine which type of synchrony is both necessary and sufficient for solving (effective or strict) termination detection. Finally, we display our conclusions and outline directions for future research in Section 5.

2 Model and Problem Definitions

2.1 System Model

We assume an asynchronous distributed system consisting of n processes, which communicate with each other by exchanging messages over a set of c communication channels. There is no global clock or shared memory.

Processes are not reliable and may fail by crashing. Once a process crashes, it halts all its operations and never recovers.

We use the terms "non-crashed process", "live process" and "operational process" interchangeably. A process that crashes is called *faulty*. A process that is never faulty is called *correct*. Note that there is a difference between the terms "live process" and "correct process". A live process has not crashed yet but may crash in the future.

Let $P = \{p_1, p_2, \ldots, p_n\}$ denote the initial set of processes in the system. We assume f to be the actual number of processes that fail during an execution, and that there is at least one correct process in the system at all times.

We assume that all channels are bidirectional but may not be FIFO (first-in-first-out). Channels are reliable in the sense that if a process never crashes, then every message destined for it is eventually delivered. A message may, however, take an arbitrary amount of time to reach its destination.

We assume the existence of a *perfect failure detector* [4], a device which gives processes reliable information about the operational state of other processes. Upon querying the local failure detector, a process receives a list of *currently suspected* processes. A perfect failure detector satisfies two properties [4]: *strong accuracy* (no process is suspected before it crashes) and *strong completeness* (a crashed process is eventually suspected by every correct process). (Although a perfect failure detector is traditionally defined for a fully connected topology, the definition can be easily extended to an arbitrary topology.)

2.2 Termination Detection in a Crash-Prone System

Informally, the termination detection problem involves determining when a distributed computation has ceased all its activity. The distributed computation satisfies the following four properties or rules. First, a process is either *active* or *passive*. Second, a process can send a message only if it is active. Third, an active process may become passive at any time. Fourth, a passive process may become active only on receiving a message. Intuitively, an active process is involved in some local activity, whereas a passive process is idle. In case both processes and channels are reliable, a distributed computation terminates once all processes become passive and stay passive thereafter. In other words, a distributed computation is said to be *classically-terminated* once all processes become passive and all channels become empty.

In a crash-prone distributed system, once a process crashes, it ceases all its activities. Moreover, any message in-transit towards a crashed process can be ignored because the message cannot initiate any new activity. Therefore, a crash-prone distributed system is said to be *strictly-terminated* if all live processes are passive and no channel contains a message in-transit towards a live process. Wu *et al* [33] establish that, for the strict-termination detection problem to be solvable in a crash-prone distributed system, it must be possible to *flush* the channel from a crashed process to a live process. A channel can be flushed using either *return-flush* [31] or *fail-flush* [18] primitive. Both primitives allow a live process to ascertain that its incoming channel from the crashed process has become empty.

In case neither return-flush nor fail-flush primitive is available, Tseng suggested *freezing* the channel from a crashed process to a live process [30]. When a live process freezes its channel with a crashed process, any message that arrives after the channel has been frozen is ignored. (A process can freeze a channel only after detecting that the process at the other end of the channel has crashed.) We say that a message is *deliverable* if it is destined for a live process along a channel that has not been frozen yet; otherwise it is *undeliverable*. We say that the system is *effectively-terminated* if all live processes are passive and there is no deliverable message in-transit towards a live process. Trivially, strict-termination implies effective-termination but not vice versa. Deciding which of the two termination conditions is to be detected depends on the application semantics.

For convenience, we refer to messages exchanged by the underlying distributed computation as *application messages* and to messages exchanged by the termination detection algorithm as *control messages*. The performance of a termination detection algorithm is measured in terms of three metrics: message complexity, detection latency and message overhead. Message complexity refers to the number of control messages exchanged by the termination detection algorithm in order to detect termination. Detection latency measures the time elapsed between when the underlying computation terminates and when the termination detection algorithm actually announces termination. Finally, message overhead refers to the amount of control data piggybacked on a message by the termination detection algorithm.

We call a termination detection algorithm *fault-tolerant* if it works correctly even in the presence of faults; otherwise it is called *fault-sensitive* or *fault-intolerant*. In this paper, we use the terms "crash", "fault" and "failure" interchangeably. Therefore, for example, the phrase "crash-tolerant" has the same meaning as the phrase "fault-tolerant".

3 From Fault-Sensitive Algorithm to Wait-Free Algorithm

We assume that the given fault-sensitive termination detection algorithm is able to detect termination of a non-diffusing computation, when any subset of processes can be initially active. This is not a restrictive assumption as it is proved in [24] that any termination detection algorithm for a diffusing computation, when at most one process is initially active, can be efficiently transformed into a termination detection algorithm for a non-diffusing computation. The transformation increases the message complexity of the underlying termination detection algorithm by only $O(n)$ messages and, moreover, does not increase its detection latency.

We also assume that, as soon as a process learns about the failure of its neighbouring process, it freezes its incoming channel with the process.

3.1 Reduction for Fully Connected Topologies

The main idea behind our transformation is to *restart* the fault-sensitive termination detection algorithm algorithm on the set of *currently operational processes* whenever a new failure is detected.

Let $\mathcal{A}$ refer to the fault-sensitive termination detection algorithm that is input to our transformation, and let $\mathcal{B}$ refer to the fault-tolerant termination detection algorithm that is outputted by our transformation.

Before restarting $\mathcal{A}$, we ensure that all operational processes agree on the set of processes that have failed. This is useful as explained further.

Consider a subset of processes Q. We say that a distributed computation has *terminated with respect to* Q (classically or strictly or effectively) if the respective termination condition holds when evaluated only on processes and channels in the subsystem induced by Q. Also, we say that Q has become *safe* if (1) all processes in $P \setminus Q$ have failed, and (2) every process in Q has learned about the failure of all processes in $P \setminus Q$. We have,

Theorem 1. *Consider a safe subset of processes Q. Assume that all processes in Q stay operational. Then a distributed computation has effectively-terminated with respect to P if and only if it has classically-terminated with respect to Q.*

Proof. (if) Assume that the distributed computation has classically-terminated with respect to Q. Therefore all processes in Q are passive and all channels among processes in Q are empty. Since Q is a safe subset of processes, all processes in $P \setminus Q$ have crashed. In other words, all live processes in the system, namely the processes in Q, are passive. Furthermore, since every process in Q knows that all processes in $P \setminus Q$ have crashed, all channels from processes in $P \setminus Q$ to processes in Q have been frozen. As a result, there is no deliverable message in transit to any process in Q. In other words, there is no deliverable message in transit to any live process in the system. Thus the distributed computation has effectively-terminated with respect to P.

(only if) Now, assume that the distributed computation has effectively-terminated with respect to P. Therefore all live processes are passive, which implies that all processes in Q are passive. Further, there is no deliverable message in transit towards any live process. Specifically, since all processes in Q are live and all processes in $P \setminus Q$ have crashed, none of the channels among processes in Q contain a deliverable message in transit. This, in turn, implies that all channels among processes in Q are actually empty. In other words, the distributed computation has classically-terminated with respect to Q.

□

The above theorem implies that if all alive processes agree on the set of failed processes and there are no further crashes, then it is sufficient to ascertain that the underlying computation has classically-terminated with respect to the set of operational processes. An advantage of detecting classical termination is that we can use $\mathcal{A}$, a fault-sensitive termination detection algorithm, to detect termination. We next show that even if one or more processes crash, $\mathcal{A}$ does not announce false termination.

Theorem 2. *When a fault-sensitive termination detection algorithm is executed on a distributed system prone to process crashes then the algorithm still satisfies the safety property, that is, it never announces false termination.*

Proof. Consider an execution γ of the distributed system in which one or more processes fail by crashing. Suppose some process p_i announces termination at time t. We show that the underlying computation has actually terminated. We construct an execution σ that is similar to γ in all respects except that the processes that have failed in γ stay operational in σ. However, they become very slow and do not execute any further steps in σ after executing their last step in γ. Moreover, in transit messages destined for such processes in γ are delayed and are received only after time t. Note that this can be done since the system is asynchronous and the termination detection algorithm is fault-sensitive (and cannot infer any information by using, say, a failure detector). Clearly, process p_i cannot distinguish between scenarios γ and σ. Therefore if p_i announces termination in γ at time t, then it also announces termination in σ at the same time. Since the termination detection algorithm works correctly for σ—a fault-free scenario—the underlying computation has indeed terminated in σ. This in turn implies that the underlying computation has terminated in γ as well. $\square$

Now, when $\mathcal{A}$ is restarted, a mechanism is needed to deal with application messages that were sent before $\mathcal{A}$ is restarted but are received after $\mathcal{A}$ has been restarted. Such application messages are referred to as stale or *old application messages*. Clearly, the current instance of $\mathcal{A}$ may not be able to handle an old application message correctly. One simple approach is to "hide" an old application message from the current instance of $\mathcal{A}$ and deliver it directly to the underlying distributed computation. However, on receiving an old application message, if the destination process changes its state from passive to active, then, to the current instance of $\mathcal{A}$, it would appear as if the process became active spontaneously. This violates one of the four rules of the distributed computation. Clearly, the current instance of $\mathcal{A}$ may not work correctly in the presence of old application messages and therefore *cannot be directly* used to detect termination of the underlying computation.

We use the following approach to deal with old application messages. We *superimpose* another computation on top of the underlying computation. We refer to the superimposed computation as the *secondary computation* and to the underlying computation as the *primary computation*. As far as live processes are concerned, the secondary computation is almost identical to the primary computation except possibly in the beginning. Whenever a process crashes and all live processes agree on the set of failed processes, we *simulate a new instance of the secondary computation* in the subsystem induced by the set of operational processes. The processes in the subsystem are referred to as the *base set* of the simulated secondary computation. We then use a *new instance of the fault-sensitive termination detection algorithm* to detect termination of the secondary computation. The older instances of the secondary computation and the fault-sensitive termination detection algorithm are simply aborted. We maintain the following invariants. First, if the secondary computation has classically terminated then the primary computation has classically terminated as well. Second, if the primary computation has classically terminated, then the secondary computation classically terminates eventually. Note that the new instances of both the secondary computation and the fault-sensitive termination detection algorithm start at the same time on the same set of processes.

We now describe the behavior of a process with respect to the secondary computation. Intuitively, a process stays active with respect to the secondary computation at least until it knows that it cannot receive any old application message in the future. Consider a safe subset of processes Q. Suppose an instance of the secondary computation is initiated in the subsystem induced by Q. A process $p_i \in Q$ is passive with respect to the current instance of the secondary computation if one of the following conditions hold:

1. it is passive with respect to the primary computation, and
2. it knows that there is no old application message in transit towards it from any process in Q

An old application message is delivered directly to the primary computation and is hidden from the current instance of the secondary computation as well as the current instance of the fault-sensitive termination detection algorithm. Specifically, only those application messages that are sent by the current instance of the secondary computation are tracked by the corresponding instance of the fault-sensitive termination detection algorithm. It can be verified that the secondary computation is "legal" in the sense that it satisfies all the four rules of the distributed computation. Therefore the fault-sensitive termination detection algorithm $\mathcal{A}$ can be safely used to detect (classical) termination of the secondary computation even in the presence of old application messages. First, we show that, to detect termination of the primary computation, it is safe to detect termination of the secondary computation.

Theorem 3. *Consider a secondary computation initiated in the subsystem induced by processes in Q. Then, if the secondary computation has classically terminated with respect to Q, then the primary computation has classically terminated with respect to Q.*

Proof. Assume that the secondary computation has classically terminated with respect to Q. Therefore all processes in Q are passive with respect to the secondary computation and no channel between processes in Q contains an application message belonging to the current instance of the secondary computation. This, in turn, implies that all processes in Q are passive with respect to the primary computation and no channel between processes in Q contains an application message belonging to the current or an older instance of the secondary computation. Moreover, since all processes in Q are passive, no process in Q has crashed, which implies that no new instance of the secondary computation has been started. Therefore the primary computation has classically terminated with respect to Q. □

Next, we prove that, to detect termination of the primary computation, it is live to detect the termination of the secondary computation under certain conditions.

Theorem 4. *Consider a secondary computation initiated in the subsystem induced by processes in Q. Assume that the primary computation has classically terminated with respect to Q and each process in Q eventually learns that there are there are no old application messages in transit towards it sent by other processes in Q. If all processes in Q stay operational, then the secondary computation eventually classically terminates with respect to Q.*

Proof. Assume that the primary computation has classically terminated with respect to Q and each process in Q eventually learns that there are there are no old application messages in transit towards it sent by other processes in Q. Clearly, since no process in Q crashes, all processes in Q eventually turn passive with respect to the secondary computation initiated on Q. Further, none of channels among processes in Q contains an application message belonging to the secondary computation initiated on Q. Therefore the secondary computation eventually classically terminates with respect to Q. □

We next describe how to ensure that all operational processes agree on the set of failed processes before restarting the secondary computation the fault-sensitive termination detection algorithm. Later, we describe how to ascertain that there are no relevant old application messages in transit. We assume that both application and control messages are piggybacked with the complement of the base set of the current instance of the secondary computation in progress, which can be used to identify the specific instance of the secondary computation.

Achieving Agreement on the Set of Failed Processes Whenever a process crashes, one of the live processes is chosen to act as the *coordinator*. Specifically, the process with the smallest identifier among all live processes acts as the coordinator. Every process, on detecting a new failure, sends a NOTIFY message to the coordinator containing the set of all processes that it knows have failed. The coordinator maintains, for each operational process p_i, processes that have failed according to p_i. On determining that all operational processes agree on the set of failed processes, the coordinator sends a RESTART message to each operational process. A RESTART message instructs a process to initiate a new instance of the secondary computation on the appropriate set of processes, and, also, start a new instance of the fault-sensitive termination detection algorithm to detect its termination.

It is possible that, before receiving a RESTART message for a new instance, a process receives an application message that is sent by a more recent instance of the secondary computation than that of the secondary computation currently in progress at that process. In that case, before processing the application message, it behaves as if it has also received a RESTART message and acts accordingly.

Tracking Old Application Messages A process stays active with respect to the current instance of the secondary computation at least until it knows that it cannot receive any old application message from one of the processes in the relevant subsystem. To that end, each process maintains a count of the number of application messages it has sent to each process so far and, also, a count of the number of application messages it has received from each process so far.

A process, on starting a new instance of the secondary computation, sends an INSTATE message to all live processes. An INSTATE message sent to process p_i contains the number of application messages sent to p_i before the process started the current instance of the secondary computation.

Clearly, once a process has received an INSTATE message from the coordinator, it can determine how many old application messages are in transit towards it and at least wait until it has received all those messages before becoming passive for the first time with respect to the current instance of the secondary computation.

3.2 Formal Description for Fully Connected Topologies

A formal description of the transformation appears in Figures 1 to 3.

3.3 Proof of Correctness for Fully Connected Topologies

We now prove that our transformation produces an algorithm $\mathcal{B}$ that solves the effective-termination detection problem given that $\mathcal{A}$ is a correct fault-sensitive algorithm for solving the classical termination detection problem.

The following proposition can be easily verified:

Proposition 1. *Whenever an instance of $\mathcal{A}$ is initiated on a process set Q, all processes in $P \setminus Q$ have in fact crashed and all channels from processes in $P \setminus Q$ to Q have been frozen.*

Transformation for process p_i:

Variables:

$failed_i$: set of process that have failed;
$coordinator_i$: process acting as the coordinator;
$current_i$: the current instance of the secondary computation;

$sent_i$: vector $[1..n]$ of number of application messages that have been sent to each process in the current instance so far;
$received_i$: vector $[1..n]$ of number of application messages that have been received from each process so far that belong to the current instance;
$oldSent_i$: vector $[1..n]$ of number of old application messages that were sent to each process in all previous instances combined;
$oldReceived_i$: vector $[1..n]$ of total number of old application messages that have been received from each process so far;
$initialized_i$: whether p_i has received the INSTATE message for the current instance;
// $othersOldSent_i$ has a valid value only if $initialized_i$ is true
$othersOldSent_i$: vector $[1..n]$ of total number of old application messages that were sent to process p_i by each process;
// $othersOldSent_i[j] - oldReceived_i[j]$ captures the number of old application messages
// sent by process p_j in transit towards process p_i

(A0) Initial action:

// initialize all variables
$current_i := \emptyset$;
$failed_i := \emptyset$;
$coordinator_i := p_1$;
$\forall k : sent_i[k] := 0$;
$\forall k : received_i[k] := 0$;
$\forall k : oldSent_i[k] := 0$:
$\forall k : oldReceived_i[k] := 0$;
$\forall k : othersOldSent_i[k] := 0$;
$initialized_i :=$ true;
call startNewInstance($current_i$);

(A1) On detecting the failure of process p_j:

// update the list of failed processes
$failed_i := failed_i \cup \{p_j\}$;
// select a new coordinator if required
$coordinator_i := \min\{p \mid p \in P$ and $p \notin failed_i\}$;
send NOTIFY($failed_i$) message to $coordinator_i$;
// all subsequent messages received from process p_i will be dropped
freeze the incoming channel from process p_j;

(A2) On receiving RESTART($instance$) from process p_j:

if $current_i \subset instance$ **then**
  // start a new instance of the secondary computation and
  // the fault-sensitive termination detection algorithm
  call startNewInstance($instance$);
endif;

Fig. 1. Transforming a fault-sensitive termination detection algorithm $\mathcal{A}$ into a fault-tolerant termination detection algorithm $\mathcal{B}$.

Transformation for process p_i (continued):

(A3) On receiving INSTATE($instance$, $othersOldSent$) from process p_j:
    if $instance = current_i$ **then**
        // *can now initialize* $othersOldsent_i$
        $othersOldSent_i := othersOldSent$;
        $initialized_i :=$ true;
    endif;

(A4) On sending an application message m to process p_j:
    $++sent_i[j]$;
    // *inform the fault-sensitive termination detection algorithm about the application message*
    $\mathcal{A}(current_i).sndApplMsg(m, p_j)$;

(A5) On receiving an application message $m(instance, controlData)$ from process p_j:
    if $instance \subset current_i$ **then**
        // *it is an old application message*
        $++oldReceived_i[j]$;
        deliver m to the underlying computation;
    else
        if $current_i \subset instance$ **then**
            // *process* p_i *has already started a new instance of the secondary computation*
            call startNewInstance($instance$);
        endif;
        $++received_i[j]$;
        // *inform the fault-sensitive termination detection algorithm about the application message*
        $\mathcal{A}(current_i).rcvAppMsg(m(controlData), p_j)$;
    endif;

(A6) On receiving a control message $m(instance)$ from process p_j:
    if $current_i \subseteq instance$ **then**
        if $current_i \subset instance$ **then**
            // *process* p_i *has already started a new instance of the secondary computation*
            call startNewInstance($instance$);
        endif;
        // *inform the fault-sensitive termination detection algorithm about the control message*
        $\mathcal{A}(current_i).rcvCtlMsg(m, p_j)$;
    endif;

(A7) On invocation of startNewInstance($instance$):
    abort $\mathcal{A}(current_i)$ and $\mathcal{SC}(current_i)$, if any;
    $current_i := instance$;
    $initialized_i :=$ false;
    $\forall k : oldSent_i[k] := oldSent_i[k] + sent_i[k]$;
    $\forall k : sent_i[k] := 0$;
    $\forall k : oldReceived_i[k] := oldReceived_i[k] + received_i[k]$;
    $\forall k : received_i[k] := 0$;
    start new instances of $\mathcal{SC}$ and $\mathcal{A}$ on $P \setminus current_i$;
        process p_i in $\mathcal{SC}$ is passive if and only if:
            (1) p_i is passive in the underlying computation, and
            (2) $initialize_i$ holds and $othersOldSent_i = oldReceived_i$
    send OUTSTATE($current_i$,$oldSent_i$) to the coordinator;

Fig. 2. Transforming a fault-sensitive termination detection algorithm $\mathcal{A}$ into a fault-tolerant termination detection algorithm $\mathcal{B}$ (continued).

Actions when process p_i becomes the coordinator:

Variables:

$othersFailed_i$: vector $[1..n]$ of set of failed processes according to each process;
$allFailed_i$: set of processes suspected by at least one process;
$instance_i$: the current instance of the secondary computation;
$toReceive_i$: number of OUTSTATE messages still to be received;
$outState_i$: vector $[1..n]$ of number of old application messages that each process has sent to other processes;

(B1) On becoming the coordinator:

```
// initialize all variables
∀k : k ≠ i : othersFailed_i[k] := ∅;
othersFailed_i[i] := failed_i;
instance_i := failed_i;
allFailed_i := failed_i;
```

(B2) On receiving NOTIFY($failed$) from process p_j:

```
// is it a new notification message?
if othersFailed_i[j] ⊂ failed then
    othersFailed_i[j] := failed;
    allFailed_i := allFailed_i ∪ failed;
    // do all operational processes agree on the set of failed processes?
    if ⟨∀k : p_k ∉ failed_i : othersFailed_i[k] = failed_i⟩ then
        instance_i := failed_i;
        send RESTART(instance_i) to each process p_k where p_k ∉ failed_i;
        toReceive_i := |failed_i|;
    endif;
endif;
```

(B3) On receiving OUTSTATE($instance$,$oldSent$) from process p_j:

```
if instance = allFailed_i then
    --toReceive_i;
    outState_i[j] := oldSent;
    // have all OUTSTATE messages been received?
    if toReceive_i = 0 then
        for k ∉ allFailed_i do
            // compute the number of old application messages sent to process p_k
            ∀l : l ∉ allFailed_i : inState_i[l] := outState_i[l][k];
            send INSTATE(instance_i,inState_i) to process p_k;
        endfor;
    endif;
endif;
```

Fig. 3. Transforming a fault-sensitive termination detection algorithm $\mathcal{A}$ into a fault-tolerant termination detection algorithm $\mathcal{B}$ (continued).

First, we prove the safety property.

Theorem 5 (safety property).
If $\mathcal{B}$ announces termination, then the underlying computation has effectively terminated.

Proof. Assume that $\mathcal{B}$ announces termination. This implies that some instance of $\mathcal{A}$ detected classical termination of the corresponding instance of the secondary computation run by some subset Q of processes. From Theorem 3, it follows that the underlying computation has also classically terminated with respect to Q. Finally, from Theorem 1, it follows that the underlying computation has effectively terminated with respect to P. $\square$

Next, we show that $\mathcal{B}$ is live. That is,

Theorem 6 (liveness property).
Once the underlying computation effectively terminates, $\mathcal{B}$ eventually announces termination.

Proof. We argue that once the underlying computation is effectively terminated, then eventually some instance of $\mathcal{A}$ announces termination. Assume that the underlying computation is effectively terminated and consider the point in time when the last process crashes. Our algorithm ensures that eventually a new instance of the secondary computation is initiated on the set Q of remaining live processes. Further, each operational process eventually learns, via an INSTATE message, the number of old application messages in transit towards it. Since the underlying computation has effectively terminated, from Theorem 1, it follows that the underlying computation has classically terminated with respect to Q. Further, using Proposition 1 and Theorem 4, it implies that the secondary computation initiated on Q classically terminates eventually. As a result, the corresponding instance of $\mathcal{A}$ eventually announces termination of the secondary computation on Q. $\square$

3.4 Performance Analysis for Fully Connected Topologies

Lemma 1. *The number of times $\mathcal{A}$ is restarted is bounded by f.*

Proof. A new instance of $\mathcal{A}$ is started only when a new failure occurs and, moreover, all operational processes have detected the failure. Since at most f processes can fail, $\mathcal{A}$ can be restarted at most f times. $\square$

To compute the message complexity of $\mathcal{B}$, we assume that $\mu(n, M)$ satisfies the following constraint for $k \geq 1$:

$$\sum_{i=1}^{k} \mu(n, c, M_i) \leq \mu(n, c, \sum_{i=1}^{k} M_i) + (k-1)\,\mu(n, 0) \tag{1}$$

For all existing termination detection algorithms that we are aware of, $\mu(n, c, M)$ is linear in M. It can be verified that if $\mu(n, c, M)$ is a linear function in M, then the inequality (1) indeed holds.

We categorize control messages into two kinds: control messages exchanged by different instances of $\mathcal{A}$, and control messages exchanged as a result of process crash, namely NOTIFY, RESTART, OUTSTATE and INSTATE. We refer to the former as type I control messages and the later as type II control messages.

Theorem 7 (message complexity). *The message complexity of $\mathcal{B}$ is given by $O(\mu(n, c, M) + f(n + \mu(n, c, 0)))$.*

Proof. Let M_i denote the number of application messages associated with with the i^{th} instance of $\mathcal{A}$. From Lemma 1, there are at most $f+1$ instances of $\mathcal{A}$. Therefore,

$$\sum_{i=1}^{f+1} M_i \quad = \quad M$$

From (1), the number of type I control messages is given by:

$$
\begin{aligned}
&= \sum_{i=1}^{f+1} \mu(n, M_i) \\
&\leq \mu(n, \sum_{i=1}^{f+1} M_i) + f\mu(n, 0) \\
&= \mu(n, M) + f\mu(n, 0)
\end{aligned}
$$

Also, the number type II control messages is at most $4n$ per fault (n NOTIFY, n RESTART, n OUTSTATE and n INSTATE). □

We now bound the detection latency of $\mathcal{B}$. We assume that message delay is at most one time unit. Moreover, once a process crashes, a live process detects the crash within one time unit.

Theorem 8 (detection latency). *The detection latency of $\mathcal{B}$ is given by $O(\delta(n, c, M) + f\delta(n, c, 0))$.*

Proof. Assume that the underlying computation has terminated. The worst-case scenario occurs when a process crashes just before the current instance of $\mathcal{A}$ is able to detect termination. Clearly, when a process fails, a new instance of the secondary computation is started on all operational processes within $O(1)$ time units assuming that there are no more failures—one time unit for failure detection, one time unit for the coordinator to receive all NOTIFY messages and one time unit for all live processes to receive RESTART messages. Once an instance of the secondary computation is initiated, it terminates with $O(1)$ time units as soon as every live process has received an INSTATE message from the coordinator. Once an instance of the secondary computation terminates, its termination is detected within $O(\delta(n, c, 0))$ time units. Note that $\delta(n, 0) = \Omega(1)$. Therefore after a processes fails its termination is detected within $O(\delta(n, c, 0))$ time units unless some other process has failed. It can be proved by induction that the termination detection can be delayed by at most $O(f\delta(n, c, 0))$ time units. □

We next bound the message overhead of $\mathcal{B}$. For the fault-sensitive termination detection algorithm $\mathcal{A}$, let $\alpha(n, c, M)$ and $\beta(n, c, M)$ denote its application message overhead and control message overhead, respectively, when the system has n processes and the underlying computation exchanges M application messages.

Theorem 9 (application message overhead). *The application message overhead of $\mathcal{B}$ is $O(\alpha(n, c, M) + f \log n)$.*

Proof. The additional information piggybacked on an application message is the set of failed processes, which is bounded by $O(f \log n)$. □

Finally, we bound the control message overhead of $\mathcal{B}$.

Theorem 10 (control message overhead). *The control message overhead of $\mathcal{B}$ for type I messages is $O(\beta(n, c, M) + f \log n)$. Also, the control message overhead for type II messages is $O(f \log n + n \log M)$.*

Proof. The additional information piggybacked on a control message is the set of failed processes, which is bounded by $O(f \log n)$. A type II control message contains the following information: (1) set of failed processes, and (2) count of the number of application messages sent so far to each process. The overhead due to the two is bounded by $O(f \log n)$ and $O(n \log M)$, respectively. □

We next discuss how our transformation can be generalized to an arbitrary topology.

3.5 Reduction for Arbitrarily Connected Topologies

When the communication topology is not fully connected, the crash of a process may disconnect the system. Clearly, once the system is disconnected, it not always possible for any one proces to detect termination of the entire distributed computation. Therefore we assume that process crashes do not partition the system.

We assume that the fault-sensitive termination detection algorithm is such that a process only needs to know its neighboring processes, and not the entire set of processes, to be able to execute the termination detection algorithm. To our knowledge, most of the termination detection algorithms can be easily modified to satisfy the above requirement, if they do not already satisfy it.

When the communication topology is not fully connected, many termination detection algorithms assume the availability of a spanning tree of the communication topology to work. The spanning tree may be unrooted or rooted. In the former case, any process can use the spanning tree to efficiently compute a snapshot of the system as in the case of wave based algorithms (*e.g.*, [25, 20, 8, 16, 2, 24, 23]). In the latter case, the root of the tree acts as a coordinator (*e.g.*, [5, 23]). Moreover, some termination detection algorithms assume that processes or channels are arranged in a logical ring (*e.g.*, [8, 22]). Clearly, if a spanning tree of the topology is available, a logical ring of processes or channels can be easily constructed. Specifically, a logical ring of processes can be obtained using a *pre-order traversal* of the tree with a token as shown in Figure 4.

Logical ring construction algorithm for process p_i:

Variables:
    $parent_i$: parent of process p_i in the spanning tree;
    $children_i$: set of children processes in the spanning tree;
    $visited_i$: set of children processes that have already been visited;

(A0) Initial action:
    $visited_i := \emptyset$;
    if $(parent_i = p_i)$ **and** $(children_i \neq \emptyset)$ **then**
        let p_k be some process in $children_i \setminus visited_i$;
        send TOKEN message to process p_k;
        add p_k to $visited_i$;
    endif;

(A1) On receiving TOKEN message from process p_j:
    if $children_i \neq visited_i$ **then**
        let p_k be some process in $children_i \setminus visited_i$;
        send TOKEN message to process p_k;
        add p_k to $visited_i$;
    else
        if $parent_i \neq p_i$ **then**
            send TOKEN message to $parent_i$;
        else
            ring construction has been completed;
        endif;
    endif;

Fig. 4. Constructing a logical ring of processes using a spanning tree.

The root of the spanning tree starts the logical ring construction algorithm by sending a TOKEN message to one of its children. A process, on receiving the TOKEN message from its parent, forwards the TOKEN message to all its children one-by-one, after which it sends the TOKEN message back to its parent. The algorithm finishes when the root receives the TOKEN message back and has sent the TOKEN message to all its children once. The ring is given by the order in which processes are visited by the TOKEN message. Clearly, in the logical ring obtained, each process appears at least once and there is a channel between every pair of consecutive processes. However, it is possible for a process to appear more than once in the ring. These multiple occurrences of processes do not affect the correctness of a termination detection algorithm. This is because, in the second and subsequent occurrences, a process can simply act as a *relay* and not execute any action with respect to the termination detection algorithm. Also, note that the size of the ring is bounded by $2n$ (each edge in the tree is visited exactly once in either direction). Therefore the performance of the termination detection algorithm is not adversely affected by multiple occurrences of processes in the ring. A similar algorithm can be used to construct a logical ring of channels from a spanning tree, which is used by some termination detection algorithms [22]. Again, in the logical ring obtained, a channel may appear more than once. However, the size of the ring is bounded by $O(c + n)$, which is $O(c)$ because $c \geq n - 1$.

We now describe our transformation. The main problems that need to be addressed are as follows. First, on occurrence of a failure, a new coordinator may have to be elected that ensures that all processes agree on the set of failed processes before restarting the termination detection algorithm. Second, a new spanning tree has to be constructed on the set of currently operational processes because the failure may have disconnected the current spanning tree. We solve both problems by using the spanning tree construction algorithm proposed by Awerbuch [3]. An advantage of Awerbuch's algorithm is that different processes can start the algorithm at different times. So, whenever a process learns about a new failure, it simply starts a *new instance* of the spanning tree construction algorithm. (Any old instance of the tree construction algorithm is aborted.) We differentiate between various instances of the spanning tree construction algorithm by using the process's knowledge about the failure of other processes when it starts the new instance, which is referred to as *instance identifier*. The instance identifier is piggybacked on every message exchanged by the spanning tree construction algorithm.

A process may learn about the failure of a process either through its local failure detector or through the instance identifier of a message received. Note that the former can only provide information about the failure of a neighboring process, whereas the latter can provide information about the failure of any process. In any case, on learning about a new failure, a process starts a new instance of the spanning tree construction algorithm as explained earlier.

If no more failures occur for a sufficiently long period of time, then all operational processes eventually learn about the failure of all crashed processes. Therefore, eventually, all operational processes start the same instance of the spanning tree construction algorithm and a valid spanning tree is eventually constructed. Once the tree construction algorithm terminates, the root of the tree elects itself as the coordinator and inform other live processes of the same. The coordinator then instructs all live processes to restart the fault-sensitive termination detection algorithm on its currently operational neighboring processes. When a termination detection algorithm is restarted, as described in Section 3.1, we need a mechanism to account for the old application messages. To that end, we can maintain counters to keep track of number application messages sent to and received from all neighboring processes as discussed in Section 3.1. Alternatively, in case all channels are FIFO, we can use markers to flush channels between operational processes [5, 24]. On restarting the fault-sensitive termination detection algorithm, a process sends a MARKER message to all its live neighboring processes. A process, on receiving a MARKER message along a channel, knows that there are no old application messages in transit along that channel. More details of the mechanism can be found in [24]. The advantage of the approach based on flushing the channels is low message overhead. If the approach based on counters is used for an arbitrary topology, then the message overhead can be as large as $O(f \log n + n^2 \log M)$. On the other hand, if the approach based on flushing channels is used, then the message overhead is only $O(f \log n)$. Hereafter we assume that all channels are FIFO and the approach based on flushing channels is used to account for old application messages.

3.6 Proof of Correctness for Arbitrarily Connected Topologies

The proof of correctness is very similar to that for fully connected communication topologies and therefore has been omitted.

3.7 Performance Analysis for Arbitrarily Connected Topologies

We now analyze the performance of our transformation. A single instance of Awerbuch's spanning tree construction algorithm [3] has $O(c+n \log n)$ message-complexity and $O(n)$ time-complexity. In the worst case, however, there can be nf different instances of the tree construction algorithm. A naive analysis therefore results in message-complexity of $n(c+n \log n)$ per fault, which is quite high. However, note that each process participates in at most f different instances of the tree construction algorithm. Let $\mathcal{I}$ denote the set of all instances of the spanning tree construction algorithm. For an instance $x \in \mathcal{I}$, let the set of participating processes be denoted by V_x. Also let E_x denote the set of channels incident on processes in V_x. It can be verified that the message complexity of Awerbuch's algorithm [3] for instance x is given by $O(c_x + n_x \log n_x)$, where $n_x = |V_x|$ and $c_x = |E_x|$.

Let $\mathcal{I}_i$ denote the set of instances in which process p_i participates. Also, let c_i denote the initial number of channels incident on process p_i. Then the total number of messages exchanged by all instances of the spanning tree construction algorithm is given by:

$$
\begin{aligned}
& \sum_{x \in \mathcal{I}} O(c_x + n_x \log n_x) \\
&= O(\sum_{x \in \mathcal{I}} \sum_{p_i \in V_x} (c_i + \log n_x)) \\
&\leq O(\sum_{x \in \mathcal{I}} \sum_{p_i \in V_x} (c_i + \log n)) \\
&= O(\sum_{p_i \in P} \sum_{x \in \mathcal{I}_i} (c_i + \log n)) \\
&\leq O(\sum_{p_i \in P} f(c_i + \log n)) \\
&= O(\sum_{p_i \in P} f(c_i + \log n)) \\
&= O(f(c + n \log n))
\end{aligned}
$$

Therefore the number of messages exchanged due to spanning tree (re)construction is given by $O(c + n \log n)$ messages per fault. Once a spanning tree has been constructed, at most $O(n)$ messages per fault are required to instruct all processes to restart the fault-sensitive termination detection algorithm. When the termination detection algorithm is restarted, at most $O(c)$ messages are required to flush all channels between operational processes.

Theorem 11 (message complexity). *For an arbitrary topology, the message complexity of $\mathcal{B}$ is given by $O(\mu(n, c, M) + f(c + n \log n + \mu(n, c, 0)))$.*

We now analyze the detection latency of the fault-tolerant termination detection algorithm. For ease of analysis, we use flooding to disseminate information about the failure of a process. Specifically, a process, on learning about a *new* failure, sends a FAILED message to all its neighboring processes. This allows operational processes to come to an agreement on the set of failed processes as soon as possible. Clearly, the number of FAILED messages is bounded by $O(c)$ per fault. Therefore message complexity stays the same. We have,

Theorem 12 (detection latency). *For an arbitrary topology, the detection latency of $\mathcal{B}$ is given by $O(\delta(n, c, M) + f(n + \delta(n, c, 0)))$.*

Proof. Assume that the underlying computation has terminated. The worst-case scenario occurs when a process crashes just before the current instance of $\mathcal{A}$ is able to detect termination. Note that once a process fails, a new instance of the secondary computation is started on all processes within $O(n)$ time units assuming that there are no more failures—$O(n)$ time units for spanning tree construction and $O(n)$ time units for all processes to receive RESTART messages from the coordinator. Once the secondary computation is initiated on all processes, it terminates within $O(1)$ time units [24]. Once the secondary computation terminates, its termination is detected within $O(\delta(n, c, 0))$ time units. Therefore after a processes fails, its termination is detected within $O(n + \delta(n, c, 0))$ time units unless some other process has failed. It can be proved by induction that the termination detection can be delayed by at most $O(f(n + \delta(n, c, 0)))$ time units. □

Theorem 13 (message overhead). *For an arbitrary topology, the application message overhead of $\mathcal{B}$ is given by $O(\alpha(n, c, M) + f \log n)$. Its control message overhead for type I messages is given by $O(\beta(n, c, M) + f \log n)$. Finally, the control message overhead for type II messages is given by $O(f \log n)$.*

3.8 Message Omissions

If channels are FIFO, the following procedure guarantees its reliability. Suppose that messages from x to a neighbour process y are always numbered. A process x should always keep recorded the number of the last message that its neighbour y acknowledged to have received.

A process x will always periodically resend the last message sent to y while its number does not match the number of the last message that its neighbour y acknowledged to have received, and that it is now recorded at x. On the other hand, only when receiving a message, y sends x either the number of the last message received from x or a warning that it knows that messages were lost together with the number of the last message received when losses started.

As the message omission is not permanent, if a message is lost, x will keep on resending the last message and y will get it at some point and notice that there is a gap in the numbers, and it will warn x. Note that x will keep on resending the last message until the number recorded at it matches, so even it the warning message is lost, at some point y will receive again this last message and send a new warning message, and so on, until x gets it and it starts resending messages from the moment there were losses.

While there are no losses (x's recorded number matches the one of the y's last message), neither x or$y$ generate extra messages.

This procedure is expensive, though. Note also that, for permanent message omissions, it would be necessary to include some device to detect them as an extra assumption.

4 Weakest Synchrony Assumption for Termination Detection

Failure detectors are not only an abstraction to yield information about the operational state of processes, they can also be regarded as *synchrony abstractions* since they are usually implemented using heartbeat messages and timeouts. For example, since an eventually perfect failure detector is strictly weaker than a perfect failure detector, it can be implemented with less synchrony assumptions (namely those of *partial synchrony* [10] instead of full synchrony). Proving that a certain type of failure detector is *necessary* for a problem gives an indication about the minimal amount of synchrony which is needed to solve that problem. Unless otherwise stated, in this section, "termination" refers to "effective-termination".

We now show that a perfect failure detector is necessary for solving termination detection in crash-prone systems. We show this by transforming an instance of the termination detection problem into a perfect failure detector at one process q, that is, q is able to reliably detect process crashes. A perfect failure detector can then be implemented by using n parallel instances of the transformation algorithm, one per process.

Assume we are given an algorithm $\mathcal{A}$ which solves termination detection for an arbitrary computation among n processes. We now set up n independent computations C_i, one per process p_i. The computation C_i is such that process p_i is initially active and all processes apart from p_i are passive. In the computation no messages are sent and received and p_i never becomes passive. Now consider some process $q \neq p_i$ and the corresponding computation C_i. Process q starts an instance of algorithm $\mathcal{A}$ with respect to computation C_i. Whenever $\mathcal{A}$ announces the termination of C_i, q henceforth permanently suspects p_i. The same actions are performed for every other process in the system, that is, q invokes n parallel instances of $\mathcal{A}$, one per computation C_i.

We now show that this algorithm implements a perfect failure detector if $\mathcal{A}$ solves termination detection. First consider strong accuracy (a process is never suspected before it crashes) and assume that q has suspected p_i. From our transformation algorithm, it follows that $\mathcal{A}$ has announced termination of the computation C_i. This means that all processes in C_i (that is, process p_i) are either crashed or passive. Since C_i is such that p_i is never passive, this implies that p_i must be crashed.

Now consider strong completeness (eventually every crashed process is suspected by every correct process) and assume that p_i crashed and q is correct. Once p_i crashes, the termination condition holds for computation C_i. Because $\mathcal{A}$ is a correct termination detection algorithm, $\mathcal{A}$ must eventually announce termination of C_i at q. Upon announcing termination, q suspects p_i, concluding the proof.

Overall, this shows that if you can solve termination detection, then you can also implement a perfect failure detector. Hence, it is impossible to solve termination detection assuming merely a failure detector which is strictly weaker than the perfect one. Therefore, the perfect failure detector is *necessary* for solving termination detection.

The *weakest failure detector* for a problem is a failure detector that is necessary and sufficient to solve that problem. Above we show that a perfect failure detector is necessary. Our transformation algorithm in Section 3 shows that a perfect failure detector is also sufficient. Therefore, perfect failure detector is the weakest failure detector for solving the effective-termination detection problem. The result holds as long as at least one process can crash and assuming that channels can be frozen. Therefore, it generalizes the result of Wu *et al* [33] which shows that a failure detector must be complete. Our result also further clarifies the relationship between the termination detection problem and the *consensus problem*: Wu *et al* [33] show that consensus is at least as hard to solve as termination detection. By relating termination detection to the failure detector hierarchy of Chandra and Toueg [4], our result has two interesting corollaries. First, termination detection is strictly harder than consensus in environments where a majority of processes remains correct. This follows from the result that in such cases the weakest failure detector for consensus is strictly weaker than the perfect failure detector [4]. Second, when any number of processes can crash, termination detection is equivalent to consensus [7].

Now,recalling the result of Wu *et al* [33] that, for the strict-termination detection problem to be solvable in a crash-prone distributed system, it must be possible to *flush* the channel from a crashed process to a correct process, we ask the following question: What is the weakest failure detector for strict-termination detection? An answer to this question sheds some light on the implementability of the flush channel abstraction, since any failure detector which can be used to solve strict-termination detection must also be sufficient to implement a flush channel.

We now prove that even a perfect failure detector cannot help to implement strict-termination detection. By the result of Wu *et al.* [33], this implies that a stronger than perfect failure detector is necessary to implement a flush channel. But, unfortunately, there is no stronger failure detector [4]. Briefly spoken, this is because a failure detector only gives information about process synchrony and not channel synchrony.

We prove the above result by showing that strict-termination detection is equivalent to a *failure detection sequencer* [13], a device which can also express channel synchrony. Roughly speaking, a failure detection sequencer, denoted Σ, behaves like a perfect failure detector if there are no crashes. However, if there is a crash, Σ not only indicates which process has crashed but also the local state of the process (the contents of all its variables) at the moment when it crashed. Note that Σ is not a failure detector in the sense of Chandra and Toueg [4] but, in a precise sense, strictly stronger, since it gives more information about the failures in the system than a perfect failure detector.

First we argue that given Σ, we can implement strict-termination detection.

We instruct every process in the system to keep track of the number of messages it has sent to and received from other processes, that is, we assume that this information is part of the local state of every process. Periodically, every correct process uses causal broadcast to send its local state to every other process. From the collected information a process can construct a consistent snapshot of the global state of the system. By looking at this snapshot, in particular the message counters, a process can determine whether all processes are passive and all channels are empty. If a snapshot satisfies this property, the process announces termination. In case there are no crashes, this scheme obviously solves strict-termination detection. In there are crashes the problem is to determine whether a channel from a crashed process towards a correct process contains in-transit messages. This is easily verified by querying Σ and looking at the final state of the crashed process. In particular, it is possible to determine whether the crashed process has sent a message which has not yet been received by some correct process. When verifying that all such messages have been received and all processes are either passive or crashed, a process announces termination. Since the information given by Σ is reliable, this in fact implies that the system has gone through a state in which the strict-termination condition holds. Hence, the algorithm solves strict-termination detection.

Now we prove that any algorithm A which solves strict-termination detection can be transformed into Σ. Again we prove this by constructing the part of Σ by which a correct process q monitors the operational state of some other process p. By composing n parallel instances of this scheme, one per process, q can monitor the operational state of *all* other processes. Again by invoking n

parallel instances of the composed scheme, every process can monitor the operational state of any other process, as required by Σ.

So assume p and q are processes and q is correct. We instruct p and q to execute the following computation C:

- p is always active.
- With every change of the variables of p, p sends a message about the state change to q.
- q is "always" passive, that is, upon receiving a message from p, q is briefly active but immediately becomes passive again.

Process q now invokes the algorithm A to detect the strict-termination of computation C. Additionally, it keeps track of the state changes of p which p regularly sends within C. The output of Σ is now emulated as follows: p is initially not suspected. When A announces termination, q suspects p permanently. Additionally, the most recent state of p is output.

We now argue that this scheme implements Σ. By the same argument as in Section 4 we can argue that the scheme implements a perfect failure detector: if A announces termination and because of the nature of C (p always active and q always passive), this implies that p must in fact have crashed. What is left to prove is to show that the state which is returned by our scheme is in fact the final state of process p, that is, the state from the moment it crashed. Since A has announced strict-termination, we know that all channels towards correct processes contain no in-transit messages. From the nature of C (whenever p has a change of its local state it sends a message), this implies that all messages from p to q have been received. Hence, there is no "more recent" state change of p which q does not know of. Therefore, the state returned by q as the output of Σ is in fact the final state of p, which concludes the proof.

Note that these results hold for strict-termination in a purely asynchronous system for any number of faulty processes. Since strict-termination needs a flush channel primitive [33], the above result implies that a flush channel needs Σ to be implemented. Since Σ is strictly stronger than a perfect failure detector, we conclude that a flush channel cannot be implemented with a perfect failure detector.

5 Conclusions and Future Work

In this paper, we present a transformation that can be used to convert any termination detection algorithm for a fully connected communication topology that has been designed for a failure-free environment into a termination detection algorithm that can tolerate process crashes. Our transformation does not impose any additional overhead on the system (besides that imposed by the underlying termination detection algorithm) if no process actually crashes during an execution. Moreover, when applied to fault-sensitive termination detection algorithms by Dijkstra and Scholten [9] and Huang [15], the resulting fault-tolerant termination detection algorithms compare *very favorably* with those by Lai and Wu [18] and Tseng [30]. Our transformation can be generalized to an arbitrary communication topology provided process crashes do not partition the system. We also prove that perfect failure detector is the weakest failure detector for solving the termination detection problem in a crash-prone distributed system.

As part of future work, we plan to investigate the termination detection problem when crashed processes may recover and channels may be lossy. We also plan to apply ideas proposed in this paper to transform other fault-sensitive algorithms—such as for detecting other stable properties—into fault-tolerant algorithms.

References

1. A. Arora and M. G. Gouda. Distributed Reset. *IEEE Transactions on Computers*, 43(9):1026–1038, 1994.
2. R. Atreya, N. Mittal, and V. K. Garg. Detecting Locally Stable Predicates without Modifying Application Messages. In *Proceedings of the 7th International Conference on Principles of Distributed Systems (OPODIS)*, La Martinique, France, December 2003.
3. B. Awerbuch. Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election, and Related Problems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, New York City, New York, May 1987.
4. T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, 1996.
5. S. Chandrasekaran and S. Venkatesan. A Message-Optimal Algorithm for Distributed Termination Detection. *Journal of Parallel and Distributed Computing (JPDC)*, 8(3):245–252, March 1990.
6. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.
7. C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The Weakest Failure Detector to Solve Certain Fundamental Problems in Distributed Computing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, St. Johns, Newfoundland, Canada, July 2004.
8. E. W. Dijkstra. Shmuel Safra's Version of Termination Detection. EWD Manuscript 998. Available at `http://www.cs.utexas.edu/users/EWD`, 1987.
9. E. W. Dijkstra and C. S. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters (IPL)*, 11(1):1–4, 1980.
10. C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.
11. N. Francez. Distributed Termination. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):42–55, January 1980.
12. F. C. Gärtner and S. Pleisch. (Im)Possibilities of Predicate Detection in Crash-Affected Systems. In *Proceedings of the 5th Workshop on Self-Stabilizing Systems (WSS)*, Lecture Notes in Computer Science (LNCS) 2194, pages 98–113, Lisbon, Portugal, October 2001. Springer-Verlag.
13. F. C. Gärtner and S. Pleisch. Failure Detection Sequencers: Necessary and Sufficient Information about Failures to Solve Predicate Detection. In *Proceedings of the 16th Symposium on Distributed Computing (DISC)*, number 2508 in Lecture Notes in Computer Science (LNCS), pages 280–294. Springer-Verlag, 2002.
14. M. Herlihy and N. Shavit. A Simple Constructive Computability Theorem for Wait-Free Computation. In *Proceedings of the 26th Annual ACM Symposium on Theory of Computing (STOC)*, pages 243–252, 1994.
15. S.-T. Huang. Detecting Termination of Distributed Computations by External Agents. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 79–84, 1989.
16. S.-T. Huang. Termination Detection by using Distributed Snapshots. *Information Processing Letters (IPL)*, 32:113–119, August 1989.
17. A. A. Khokhar, S. E. Hambrusch, and E. Kocalar. Termination Detection in Data-Driven Parallel Computations/Applications. *Journal of Parallel and Distributed Computing (JPDC)*, 63(3):312–326, March 2003.
18. T.-H. Lai and L.-F. Wu. An $(N-1)$-Resilient Algorithm for Distributed Termination Detection. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 6(1):63–78, January 1995.
19. N. R. Mahapatra and S. Dutt. An Efficient Delay-Optimal Distributed Termination Detection Algorithm. To Appear in Journal of Parallel and Distributed Computing (JPDC), 2004.
20. F. Mattern. Algorithms for Distributed Termination Detection. *Distributed Computing (DC)*, 2(3):161–175, 1987.
21. F. Mattern. Global Quiescence Detection based on Credit Distribution and Recovery. *Information Processing Letters (IPL)*, 30(4):195–200, February 1989.
22. J. Misra. Detecting Termination of Distributed Computations using Markers. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)*, pages 290–294, 1983.
23. N. Mittal, S. Venkatesan, and S. Peri. Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies. In *Proceedings of the Symposium on Distributed Computing (DISC)*, Amsterdam, The Netherlands, October 2004.
24. S. Peri and N. Mittal. On Termination Detection in an Asynchronous Distributed System. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems (PDCS)*, San Francisco, California, September 2004.
25. S. P. Rana. A Distributed Solution of the Distributed Termination Problem. *Information Processing Letters (IPL)*, 17(1):43–46, 1983.
26. A. Shah and S. Toueg. Distributed Snapshots in spite of Failures. Technical Report TR84-624, Department of Computer Science, Cornell University, Ithaca, NY, July 1984. (Revised February 1985).

27. N. Shavit and N. Francez. A New Approach to Detection of Locally Indicative Stability. In *Proceedings of the International Colloquium on Automata, Languages and Systems (ICALP)*, pages 344–358, Rennes, France, 1986.
28. G. Stupp. Stateless Termination Detection. In *Proceedings of the 16th Symposium on Distributed Computing (DISC)*, pages 163–172, Toulouse, France, 2002.
29. G. Tel and F. Mattern. The Derivation of Distributed Termination Detection Algorithms from Garbage Collection Schemes. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(1):1–35, January 1993.
30. Y.-C. Tseng. Detecting Termination by Weight-Throwing in a Faulty Distributed System. *Journal of Parallel and Distributed Computing (JPDC)*, 25(1):7–15, February 1995.
31. S. Venkatesan. Reliable Protocols for Distributed Termination Detection. *IEEE Transactions on Reliability*, 38(1):103–110, April 1989.
32. X. Wang and J. Mayo. A General Model for Detecting Termination in Dynamic Systems. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*, Santa Fe, New Mexico, April 2004.
33. L.-F. Wu, T.-H. Lai, and Y.-C. Tseng. Consensus and Termination Detection in the Presence of Faulty Processes. In *Proceedings of the International Conference on Parallel and Distributed Systems (ICPADS)*, pages 267–274, Hsinchu, Taiwan, December 1992.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

1987-01 *	Fachgruppe Informatik: Jahresbericht 1986
1987-02 *	David de Frutos Escrig, Klaus Indermark: Equivalence Relations of Non-Deterministic Ianov-Schemes
1987-03 *	Manfred Nagl: A Software Development Environment based on Graph Technology
1987-04 *	Claus Lewerentz, Manfred Nagl, Bernhard Westfechtel: On Integration Mechanisms within a Graph-Based Software Development Environment
1987-05 *	Reinhard Rinn: Über Eingabeanomalien bei verschiedenen Inferenzmodellen
1987-06 *	Werner Damm, Gert Döhmen: Specifying Distributed Computer Architectures in AADL*
1987-07 *	Gregor Engels, Claus Lewerentz, Wilhelm Schäfer: Graph Grammar Engineering: A Software Specification Method
1987-08 *	Manfred Nagl: Set Theoretic Approaches to Graph Grammars
1987-09 *	Claus Lewerentz, Andreas Schürr: Experiences with a Database System for Software Documents
1987-10 *	Herbert Klaeren, Klaus Indermark: A New Implementation Technique for Recursive Function Definitions
1987-11 *	Rita Loogen: Design of a Parallel Programmable Graph Reduction Machine with Distributed Memory
1987-12	J. Börstler, U. Möncke, R. Wilhelm: Table compression for tree automata
1988-01 *	Gabriele Esser, Johannes Rückert, Frank Wagner Gesellschaftliche Aspekte der Informatik
1988-02 *	Peter Martini, Otto Spaniol: Token-Passing in High-Speed Backbone Networks for Campus-Wide Environments
1988-03 *	Thomas Welzel: Simulation of a Multiple Token Ring Backbone
1988-04 *	Peter Martini: Performance Comparison for HSLAN Media Access Protocols
1988-05 *	Peter Martini: Performance Analysis of Multiple Token Rings
1988-06 *	Andreas Mann, Johannes Rückert, Otto Spaniol: Datenfunknetze
1988-07 *	Andreas Mann, Johannes Rückert: Packet Radio Networks for Data Exchange
1988-08 *	Andreas Mann, Johannes Rückert: Concurrent Slot Assignment Protocol for Packet Radio Networks
1988-09 *	W. Kremer, F. Reichert, J. Rückert, A. Mann: Entwurf einer Netzwerktopologie für ein Mobilfunknetz zur Unterstützung des öffentlichen Straßenverkehrs
1988-10 *	Kai Jakobs: Towards User-Friendly Networking
1988-11 *	Kai Jakobs: The Directory - Evolution of a Standard
1988-12 *	Kai Jakobs: Directory Services in Distributed Systems - A Survey
1988-13 *	Martine Schümmer: RS-511, a Protocol for the Plant Floor

1988-14 * U. Quernheim: Satellite Communication Protocols - A Performance Comparison Considering On-Board Processing

1988-15 * Peter Martini, Otto Spaniol, Thomas Welzel: File Transfer in High Speed Token Ring Networks: Performance Evaluation by Approximate Analysis and Simulation

1988-16 * Fachgruppe Informatik: Jahresbericht 1987

1988-17 * Wolfgang Thomas: Automata on Infinite Objects

1988-18 * Michael Sonnenschein: On Petri Nets and Data Flow Graphs

1988-19 * Heiko Vogler: Functional Distribution of the Contextual Analysis in Block-Structured Programming Languages: A Case Study of Tree Transducers

1988-20 * Thomas Welzel: Einsatz des Simulationswerkzeuges QNAP2 zur Leistungsbewertung von Kommunikationsprotokollen

1988-21 * Th. Janning, C. Lewerentz: Integrated Project Team Management in a Software Development Environment

1988-22 * Joost Engelfriet, Heiko Vogler: Modular Tree Transducers

1988-23 * Wolfgang Thomas: Automata and Quantifier Hierarchies

1988-24 * Uschi Heuter: Generalized Definite Tree Languages

1989-01 * Fachgruppe Informatik: Jahresbericht 1988

1989-02 * G. Esser, J. Rückert, F. Wagner (Hrsg.): Gesellschaftliche Aspekte der Informatik

1989-03 * Heiko Vogler: Bottom-Up Computation of Primitive Recursive Tree Functions

1989-04 * Andy Schürr: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language

1989-05 J. Börstler: Reuse and Software Development - Problems, Solutions, and Bibliography (in German)

1989-06 * Kai Jakobs: OSI - An Appropriate Basis for Group Communication?

1989-07 * Kai Jakobs: ISO's Directory Proposal - Evolution, Current Status and Future Problems

1989-08 * Bernhard Westfechtel: Extension of a Graph Storage for Software Documents with Primitives for Undo/Redo and Revision Control

1989-09 * Peter Martini: High Speed Local Area Networks - A Tutorial

1989-10 * P. Davids, Th. Welzel: Performance Analysis of DQDB Based on Simulation

1989-11 * Manfred Nagl (Ed.): Abstracts of Talks presented at the WG '89 15th International Workshop on Graphtheoretic Concepts in Computer Science

1989-12 * Peter Martini: The DQDB Protocol - Is it Playing the Game?

1989-13 * Martine Schümmer: CNC/DNC Communication with MAP

1989-14 * Martine Schümmer: Local Area Networks for Manufactoring Environments with hard Real-Time Requirements

1989-15 * M. Schümmer, Th. Welzel, P. Martini: Integration of Field Bus and MAP Networks - Hierarchical Communication Systems in Production Environments

1989-16 * G. Vossen, K.-U. Witt: SUXESS: Towards a Sound Unification of Extensions of the Relational Data Model

1989-17 * J. Derissen, P. Hruschka, M.v.d. Beeck, Th. Janning, M. Nagl: Integrating Structured Analysis and Information Modelling

1989-18 A. Maassen: Programming with Higher Order Functions

1989-19 * Mario Rodriguez-Artalejo, Heiko Vogler: A Narrowing Machine for Syntax Directed BABEL

1989-20 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Graph-based Implementation of a Functional Logic Language

1990-01 * Fachgruppe Informatik: Jahresbericht 1989

1990-02 * Vera Jansen, Andreas Potthoff, Wolfgang Thomas, Udo Wermuth: A Short Guide to the AMORE System (Computing Automata, MOnoids and Regular Expressions)

1990-03 * Jerzy Skurczynski: On Three Hierarchies of Weak SkS Formulas

1990-04 R. Loogen: Stack-based Implementation of Narrowing

1990-05 H. Kuchen, A. Wagener: Comparison of Dynamic Load Balancing Strategies

1990-06 * Kai Jakobs, Frank Reichert: Directory Services for Mobile Communication

1990-07 * Kai Jakobs: What's Beyond the Interface - OSI Networks to Support Cooperative Work

1990-08 * Kai Jakobs: Directory Names and Schema - An Evaluation

1990-09 * Ulrich Quernheim, Dieter Kreuer: Das CCITT - Signalisierungssystem Nr. 7 auf Satellitenstrecken; Simulation der Zeichengabestrecke

1990-11 H. Kuchen, R. Loogen, J.J. Moreno Navarro, M. Rodriguez Artalejo: Lazy Narrowing in a Graph Machine

1990-12 * Kai Jakobs, Josef Kaltwasser, Frank Reichert, Otto Spaniol: Der Computer fährt mit

1990-13 * Rudolf Mathar, Andreas Mann: Analyzing a Distributed Slot Assignment Protocol by Markov Chains

1990-14 A. Maassen: Compilerentwicklung in Miranda - ein Praktikum in funktionaler Programmierung (written in german)

1990-15 * Manfred Nagl, Andreas Schürr: A Specification Environment for Graph Grammars

1990-16 A. Schürr: PROGRESS: A VHL-Language Based on Graph Grammars

1990-17 * Marita Möller: Ein Ebenenmodell wissensbasierter Konsultationen - Unterstützung für Wissensakquisition und Erklärungsfähigkeit

1990-18 * Eric Kowalewski: Entwurf und Interpretation einer Sprache zur Beschreibung von Konsultationsphasen in Expertensystemen

1990-20 Y. Ortega Mallen, D. de Frutos Escrig: A Complete Proof System for Timed Observations

1990-21 * Manfred Nagl: Modelling of Software Architectures: Importance, Notions, Experiences

1990-22 H. Fassbender, H. Vogler: A Call-by-need Implementation of Syntax Directed Functional Programming

1991-01 Guenther Geiler (ed.), Fachgruppe Informatik: Jahresbericht 1990

1991-03 B. Steffen, A. Ingolfsdottir: Characteristic Formulae for Processes with Divergence

1991-04 M. Portz: A new class of cryptosystems based on interconnection networks

1991-05 H. Kuchen, G. Geiler: Distributed Applicative Arrays
1991-06 * Ludwig Staiger: Kolmogorov Complexity and Hausdorff Dimension
1991-07 * Ludwig Staiger: Syntactic Congruences for w-languages
1991-09 * Eila Kuikka: A Proposal for a Syntax-Directed Text Processing System
1991-10 K. Gladitz, H. Fassbender, H. Vogler: Compiler-based Implementation of Syntax-Directed Functional Programming
1991-11 R. Loogen, St. Winkler: Dynamic Detection of Determinism in Functional Logic Languages
1991-12 * K. Indermark, M. Rodriguez Artalejo (Eds.): Granada Workshop on the Integration of Functional and Logic Programming
1991-13 * Rolf Hager, Wolfgang Kremer: The Adaptive Priority Scheduler: A More Fair Priority Service Discipline
1991-14 * Andreas Fasbender, Wolfgang Kremer: A New Approximation Algorithm for Tandem Networks with Priority Nodes
1991-15 J. Börstler, A. Zündorf: Revisiting extensions to Modula-2 to support reusability
1991-16 J. Börstler, Th. Janning: Bridging the gap between Requirements Analysis and Design
1991-17 A. Zündorf, A. Schürr: Nondeterministic Control Structures for Graph Rewriting Systems
1991-18 * Matthias Jarke, John Mylopoulos, Joachim W. Schmidt, Yannis Vassiliou: DAIDA: An Environment for Evolving Information Systems
1991-19 M. Jeusfeld, M. Jarke: From Relational to Object-Oriented Integrity Simplification
1991-20 G. Hogen, A. Kindler, R. Loogen: Automatic Parallelization of Lazy Functional Programs
1991-21 * Prof. Dr. rer. nat. Otto Spaniol: ODP (Open Distributed Processing): Yet another Viewpoint
1991-22 H. Kuchen, F. Lücking, H. Stoltze: The Topology Description Language TDL
1991-23 S. Graf, B. Steffen: Compositional Minimization of Finite State Systems
1991-24 R. Cleaveland, J. Parrow, B. Steffen: The Concurrency Workbench: A Semantics Based Tool for the Verification of Concurrent Systems
1991-25 * Rudolf Mathar, Jürgen Mattfeldt: Optimal Transmission Ranges for Mobile Communication in Linear Multihop Packet Radio Networks
1991-26 M. Jeusfeld, M. Staudt: Query Optimization in Deductive Object Bases
1991-27 J. Knoop, B. Steffen: The Interprocedural Coincidence Theorem
1991-28 J. Knoop, B. Steffen: Unifying Strength Reduction and Semantic Code Motion
1991-30 T. Margaria: First-Order theories for the verification of complex FSMs
1991-31 B. Steffen: Generating Data Flow Analysis Algorithms from Modal Specifications
1992-01 Stefan Eherer (ed.), Fachgruppe Informatik: Jahresbericht 1991
1992-02 * Bernhard Westfechtel: Basismechanismen zur Datenverwaltung in strukturbezogenen Hypertextsystemen
1992-04 S. A. Smolka, B. Steffen: Priority as Extremal Probability
1992-05 * Matthias Jarke, Carlos Maltzahn, Thomas Rose: Sharing Processes: Team Coordination in Design Repositories

1992-06 O. Burkart, B. Steffen: Model Checking for Context-Free Processes

1992-07 * Matthias Jarke, Klaus Pohl: Information Systems Quality and Quality Information Systems

1992-08 * Rudolf Mathar, Jürgen Mattfeldt: Analyzing Routing Strategy NFP in Multihop Packet Radio Networks on a Line

1992-09 * Alfons Kemper, Guido Moerkotte: Grundlagen objektorientierter Datenbanksysteme

1992-10 Matthias Jarke, Manfred Jeusfeld, Andreas Miethsam, Michael Gocek: Towards a logic-based reconstruction of software configuration management

1992-11 Werner Hans: A Complete Indexing Scheme for WAM-based Abstract Machines

1992-12 W. Hans, R. Loogen, St. Winkler: On the Interaction of Lazy Evaluation and Backtracking

1992-13 * Matthias Jarke, Thomas Rose: Specification Management with CAD

1992-14 Th. Noll, H. Vogler: Top-down Parsing with Simultaneous Evaluation on Noncircular Attribute Grammars

1992-15 A. Schuerr, B. Westfechtel: Graphgrammatiken und Graphersetzungssysteme(written in german)

1992-16 * Graduiertenkolleg Informatik und Technik (Hrsg.): Forschungsprojekte des Graduiertenkollegs Informatik und Technik

1992-17 M. Jarke (ed.): ConceptBase V3.1 User Manual

1992-18 * Clarence A. Ellis, Matthias Jarke (Eds.): Distributed Cooperation in Integrated Information Systems - Proceedings of the Third International Workshop on Intelligent and Cooperative Information Systems

1992-19-00 H. Kuchen, R. Loogen (eds.): Proceedings of the 4th Int. Workshop on the Parallel Implementation of Functional Languages

1992-19-01 G. Hogen, R. Loogen: PASTEL - A Parallel Stack-Based Implementation of Eager Functional Programs with Lazy Data Structures (Extended Abstract)

1992-19-02 H. Kuchen, K. Gladitz: Implementing Bags on a Shared Memory MIMD-Machine

1992-19-03 C. Rathsack, S.B. Scholz: LISA - A Lazy Interpreter for a Full-Fledged Lambda-Calculus

1992-19-04 T.A. Bratvold: Determining Useful Parallelism in Higher Order Functions

1992-19-05 S. Kahrs: Polymorphic Type Checking by Interpretation of Code

1992-19-06 M. Chakravarty, M. Köhler: Equational Constraints, Residuation, and the Parallel JUMP-Machine

1992-19-07 J. Seward: Polymorphic Strictness Analysis using Frontiers (Draft Version)

1992-19-08 D. Gärtner, A. Kimms, W. Kluge: pi-Redˆ+ - A Compiling Graph-Reduction System for a Full Fledged Lambda-Calculus

1992-19-09 D. Howe, G. Burn: Experiments with strict STG code

1992-19-10 J. Glauert: Parallel Implementation of Functional Languages Using Small Processes

1992-19-11 M. Joy, T. Axford: A Parallel Graph Reduction Machine

1992-19-12 A. Bennett, P. Kelly: Simulation of Multicache Parallel Reduction

1992-19-13 K. Langendoen, D.J. Agterkamp: Cache Behaviour of Lazy Functional Programs (Working Paper)
1992-19-14 K. Hammond, S. Peyton Jones: Profiling scheduling strategies on the GRIP parallel reducer
1992-19-15 S. Mintchev: Using Strictness Information in the STG-machine
1992-19-16 D. Rushall: An Attribute Grammar Evaluator in Haskell
1992-19-17 J. Wild, H. Glaser, P. Hartel: Statistics on storage management in a lazy functional language implementation
1992-19-18 W.S. Martins: Parallel Implementations of Functional Languages
1992-19-19 D. Lester: Distributed Garbage Collection of Cyclic Structures (Draft version)
1992-19-20 J.C. Glas, R.F.H. Hofman, W.G. Vree: Parallelization of Branch-and-Bound Algorithms in a Functional Programming Environment
1992-19-21 S. Hwang, D. Rushall: The nu-STG machine: a parallelized Spineless Tagless Graph Reduction Machine in a distributed memory architecture (Draft version)
1992-19-22 G. Burn, D. Le Metayer: Cps-Translation and the Correctness of Optimising Compilers
1992-19-23 S.L. Peyton Jones, P. Wadler: Imperative functional programming (Brief summary)
1992-19-24 W. Damm, F. Liu, Th. Peikenkamp: Evaluation and Parallelization of Functions in Functional + Logic Languages (abstract)
1992-19-25 M. Kesseler: Communication Issues Regarding Parallel Functional Graph Rewriting
1992-19-26 Th. Peikenkamp: Charakterizing and representing neededness in functional loginc languages (abstract)
1992-19-27 H. Doerr: Monitoring with Graph-Grammars as formal operational Models
1992-19-28 J. van Groningen: Some implementation aspects of Concurrent Clean on distributed memory architectures
1992-19-29 G. Ostheimer: Load Bounding for Implicit Parallelism (abstract)
1992-20 H. Kuchen, F.J. Lopez Fraguas, J.J. Moreno Navarro, M. Rodriguez Artalejo: Implementing Disequality in a Lazy Functional Logic Language
1992-21 H. Kuchen, F.J. Lopez Fraguas: Result Directed Computing in a Functional Logic Language
1992-22 H. Kuchen, J.J. Moreno Navarro, M.V. Hermenegildo: Independent AND-Parallel Narrowing
1992-23 T. Margaria, B. Steffen: Distinguishing Formulas for Free
1992-24 K. Pohl: The Three Dimensions of Requirements Engineering
1992-25 * R. Stainov: A Dynamic Configuration Facility for Multimedia Communications
1992-26 * Michael von der Beeck: Integration of Structured Analysis and Timed Statecharts for Real-Time and Concurrency Specification
1992-27 W. Hans, St. Winkler: Aliasing and Groundness Analysis of Logic Programs through Abstract Interpretation and its Safety
1992-28 * Gerhard Steinke, Matthias Jarke: Support for Security Modeling in Information Systems Design
1992-29 B. Schinzel: Warum Frauenforschung in Naturwissenschaft und Technik

1992-30 A. Kemper, G. Moerkotte, K. Peithner: Object-Orientation Axiomatised by Dynamic Logic

1992-32 * Bernd Heinrichs, Kai Jakobs: Timer Handling in High-Performance Transport Systems

1992-33 * B. Heinrichs, K. Jakobs, K. Lenßen, W. Reinhardt, A. Spinner: Euro-Bridge: Communication Services for Multimedia Applications

1992-34 C. Gerlhof, A. Kemper, Ch. Kilger, G. Moerkotte: Partition-Based Clustering in Object Bases: From Theory to Practice

1992-35 J. Börstler: Feature-Oriented Classification and Reuse in IPSEN

1992-36 M. Jarke, J. Bubenko, C. Rolland, A. Sutcliffe, Y. Vassiliou: Theories Underlying Requirements Engineering: An Overview of NATURE at Genesis

1992-37 * K. Pohl, M. Jarke: Quality Information Systems: Repository Support for Evolving Process Models

1992-38 A. Zuendorf: Implementation of the imperative / rule based language PROGRES

1992-39 P. Koch: Intelligentes Backtracking bei der Auswertung funktional-logischer Programme

1992-40 * Rudolf Mathar, Jürgen Mattfeldt: Channel Assignment in Cellular Radio Networks

1992-41 * Gerhard Friedrich, Wolfgang Neidl: Constructive Utility in Model-Based Diagnosis Repair Systems

1992-42 * P. S. Chen, R. Hennicker, M. Jarke: On the Retrieval of Reusable Software Components

1992-43 W. Hans, St.Winkler: Abstract Interpretation of Functional Logic Languages

1992-44 N. Kiesel, A. Schuerr, B. Westfechtel: Design and Evaluation of GRAS, a Graph-Oriented Database System for Engineering Applications

1993-01 * Fachgruppe Informatik: Jahresbericht 1992

1993-02 * Patrick Shicheng Chen: On Inference Rules of Logic-Based Information Retrieval Systems

1993-03 G. Hogen, R. Loogen: A New Stack Technique for the Management of Runtime Structures in Distributed Environments

1993-05 A. Zündorf: A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRES

1993-06 A. Kemper, D. Kossmann: Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis

1993-07 * Graduiertenkolleg Informatik und Technik (Hrsg.): Graduiertenkolleg Informatik und Technik

1993-08 * Matthias Berger: k-Coloring Vertices using a Neural Network with Convergence to Valid Solutions

1993-09 M. Buchheit, M. Jeusfeld, W. Nutt, M. Staudt: Subsumption between Queries to Object-Oriented Databases

1993-10 O. Burkart, B. Steffen: Pushdown Processes: Parallel Composition and Model Checking

1993-11 * R. Große-Wienker, O. Hermanns, D. Menzenbach, A. Pollacks, S. Repetzki, J. Schwartz, K. Sonnenschein, B. Westfechtel: Das SUKITS-Projekt: A-posteriori-Integration heterogener CIM-Anwendungssysteme

1993-12 * Rudolf Mathar, Jürgen Mattfeldt: On the Distribution of Cumulated Interference Power in Rayleigh Fading Channels

1993-13 O. Maler, L. Staiger: On Syntactic Congruences for omega-languages

1993-14 M. Jarke, St. Eherer, R. Gallersdoerfer, M. Jeusfeld, M. Staudt: ConceptBase - A Deductive Object Base Manager

1993-15 M. Staudt, H.W. Nissen, M.A. Jeusfeld: Query by Class, Rule and Concept

1993-16 * M. Jarke, K. Pohl, St. Jacobs et al.: Requirements Engineering: An Integrated View of Representation Process and Domain

1993-17 * M. Jarke, K. Pohl: Establishing Vision in Context: Towards a Model of Requirements Processes

1993-18 W. Hans, H. Kuchen, St. Winkler: Full Indexing for Lazy Narrowing

1993-19 W. Hans, J.J. Ruz, F. Saenz, St. Winkler: A VHDL Specification of a Shared Memory Parallel Machine for Babel

1993-20 * K. Finke, M. Jarke, P. Szczurko, R. Soltysiak: Quality Management for Expert Systems in Process Control

1993-21 M. Jarke, M.A. Jeusfeld, P. Szczurko: Three Aspects of Intelligent Cooperation in the Quality Cycle

1994-01 Margit Generet, Sven Martin (eds.), Fachgruppe Informatik: Jahresbericht 1993

1994-02 M. Lefering: Development of Incremental Integration Tools Using Formal Specifications

1994-03 * P. Constantopoulos, M. Jarke, J. Mylopoulos, Y. Vassiliou: The Software Information Base: A Server for Reuse

1994-04 * Rolf Hager, Rudolf Mathar, Jürgen Mattfeldt: Intelligent Cruise Control and Reliable Communication of Mobile Stations

1994-05 * Rolf Hager, Peter Hermesmann, Michael Portz: Feasibility of Authentication Procedures within Advanced Transport Telematics

1994-06 * Claudia Popien, Bernd Meyer, Axel Kuepper: A Formal Approach to Service Import in ODP Trader Federations

1994-07 P. Peters, P. Szczurko: Integrating Models of Quality Management Methods by an Object-Oriented Repository

1994-08 * Manfred Nagl, Bernhard Westfechtel: A Universal Component for the Administration in Distributed and Integrated Development Environments

1994-09 * Patrick Horster, Holger Petersen: Signatur- und Authentifikationsverfahren auf der Basis des diskreten Logarithmusproblems

1994-11 A. Schürr: PROGRES, A Visual Language and Environment for PROgramming with Graph REwrite Systems

1994-12 A. Schürr: Specification of Graph Translators with Triple Graph Grammars

1994-13 A. Schürr: Logic Based Programmed Structure Rewriting Systems

1994-14 L. Staiger: Codes, Simplifying Words, and Open Set Condition

1994-15 * Bernhard Westfechtel: A Graph-Based System for Managing Configurations of Engineering Design Documents

1994-16 P. Klein: Designing Software with Modula-3

1994-17 I. Litovsky, L. Staiger: Finite acceptance of infinite words

1994-18 G. Hogen, R. Loogen: Parallel Functional Implementations: Graphbased vs. Stackbased Reduction

1994-19 M. Jeusfeld, U. Johnen: An Executable Meta Model for Re-Engineering of Database Schemas

1994-20 * R. Gallersdörfer, M. Jarke, K. Klabunde: Intelligent Networks as a Data Intensive Application (INDIA)

1994-21 M. Mohnen: Proving the Correctness of the Static Link Technique Using Evolving Algebras

1994-22 H. Fernau, L. Staiger: Valuations and Unambiguity of Languages, with Applications to Fractal Geometry

1994-24 * M. Jarke, K. Pohl, R. Dömges, St. Jacobs, H. W. Nissen: Requirements Information Management: The NATURE Approach

1994-25 * M. Jarke, K. Pohl, C. Rolland, J.-R. Schmitt: Experience-Based Method Evaluation and Improvement: A Process Modeling Approach

1994-26 * St. Jacobs, St. Kethers: Improving Communication and Decision Making within Quality Function Deployment

1994-27 * M. Jarke, H. W. Nissen, K. Pohl: Tool Integration in Evolving Information Systems Environments

1994-28 O. Burkart, D. Caucal, B. Steffen: An Elementary Bisimulation Decision Procedure for Arbitrary Context-Free Processes

1995-01 * Fachgruppe Informatik: Jahresbericht 1994

1995-02 Andy Schürr, Andreas J. Winter, Albert Zündorf: Graph Grammar Engineering with PROGRES

1995-03 Ludwig Staiger: A Tight Upper Bound on Kolmogorov Complexity by Hausdorff Dimension and Uniformly Optimal Prediction

1995-04 Birgitta König-Ries, Sven Helmer, Guido Moerkotte: An experimental study on the complexity of left-deep join ordering problems for cyclic queries

1995-05 Sophie Cluet, Guido Moerkotte: Efficient Evaluation of Aggregates on Bulk Types

1995-06 Sophie Cluet, Guido Moerkotte: Nested Queries in Object Bases

1995-07 Sophie Cluet, Guido Moerkotte: Query Optimization Techniques Exploiting Class Hierarchies

1995-08 Markus Mohnen: Efficient Compile-Time Garbage Collection for Arbitrary Data Structures

1995-09 Markus Mohnen: Functional Specification of Imperative Programs: An Alternative Point of View of Functional Languages

1995-10 Rainer Gallersdörfer, Matthias Nicola: Improving Performance in Replicated Databases through Relaxed Coherency

1995-11 * M.Staudt, K.von Thadden: Subsumption Checking in Knowledge Bases

1995-12 * G.V.Zemanek, H.W.Nissen, H.Hubert, M.Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology

1995-13 * M.Staudt, M.Jarke: Incremental Maintenance of Externally Materialized Views

1995-14 * P.Peters, P.Szczurko, M.Jeusfeld: Oriented Information Management: Conceptual Models at Work

1995-15 * Matthias Jarke, Sudha Ram (Hrsg.): WITS 95 Proceedings of the 5th Annual Workshop on Information Technologies and Systems

1995-16 * W.Hans, St.Winkler, F.Saenz: Distributed Execution in Functional Logic Programming

1996-01 * Jahresbericht 1995

1996-02 Michael Hanus, Christian Prehofer: Higher-Order Narrowing with Definitional Trees

1996-03 * W.Scheufele, G.Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates

1996-04 Klaus Pohl: PRO-ART: Enabling Requirements Pre-Traceability

1996-05 Klaus Pohl: Requirements Engineering: An Overview

1996-06 * M.Jarke, W.Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools

1996-07 Olaf Chitil: The Sigma-Semantics: A Comprehensive Semantics for Functional Programs

1996-08 * S.Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics

1996-09 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP96 - Fifth International Conference on Algebraic and Logic Programming

1996-09-0 Michael Hanus (Ed.): Proceedings of the Poster Session of ALP 96 - Fifth International Conference on Algebraic and Logic Programming: Introduction and table of contents

1996-09-1 Ilies Alouini: An Implementation of Conditional Concurrent Rewriting on Distributed Memory Machines

1996-09-2 Olivier Danvy, Karoline Malmkjær: On the Idempotence of the CPS Transformation

1996-09-3 Víctor M. Gulías, José L. Freire: Concurrent Programming in Haskell

1996-09-4 Sébastien Limet, Pierre Réty: On Decidability of Unifiability Modulo Rewrite Systems

1996-09-5 Alexandre Tessier: Declarative Debugging in Constraint Logic Programming

1996-10 Reidar Conradi, Bernhard Westfechtel: Version Models for Software Configuration Management

1996-11 * C.Weise, D.Lenzkes: A Fast Decision Algorithm for Timed Refinement

1996-12 * R.Dömges, K.Pohl, M.Jarke, B.Lohmann, W.Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models

1996-13 * K.Pohl, R.Klamma, K.Weidenhaupt, R.Dömges, P.Haumer, M.Jarke: A Framework for Process-Integrated Tools

1996-14 * R.Gallersdörfer, K.Klabunde, A.Stolz, M.Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

1996-15 * H.Schimpe, M.Staudt: VAREX: An Environment for Validating and Refining Rule Bases

1996-16 * M.Jarke, M.Gebhardt, S.Jacobs, H.Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization

1996-17 Manfred A. Jeusfeld, Tung X. Bui: Decision Support Components on the Internet

1996-18 Manfred A. Jeusfeld, Mike Papazoglou: Information Brokering: Design, Search and Transformation

1996-19 * P.Peters, M.Jarke: Simulating the impact of information flows in networked organizations

1996-20 Matthias Jarke, Peter Peters, Manfred A. Jeusfeld: Model-driven planning and design of cooperative information systems

1996-21 * G.de Michelis, E.Dubois, M.Jarke, F.Matthes, J.Mylopoulos, K.Pohl, J.Schmidt, C.Woo, E.Yu: Cooperative information systems: a manifesto

1996-22 * S.Jacobs, M.Gebhardt, S.Kethers, W.Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality

1996-23 * M.Gebhardt, S.Jacobs: Conflict Management in Design

1997-01 Michael Hanus, Frank Zartmann (eds.): Jahresbericht 1996

1997-02 Johannes Faassen: Using full parallel Boltzmann Machines for Optimization

1997-03 Andreas Winter, Andy Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems

1997-04 Markus Mohnen, Stefan Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler

1997-05 * S.Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge

1997-06 Matthias Nicola, Matthias Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries

1997-07 Petra Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen

1997-08 Dorothea Blostein, Andy Schürr: Computing with Graphs and Graph Rewriting

1997-09 Carl-Arndt Krapp, Bernhard Westfechtel: Feedback Handling in Dynamic Task Nets

1997-10 Matthias Nicola, Matthias Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases

1997-11 * R. Klamma, P. Peters, M. Jarke: Workflow Support for Failure Management in Federated Organizations

1997-13 Markus Mohnen: Optimising the Memory Management of Higher-Order Functional Programs

1997-14 Roland Baumann: Client/Server Distribution in a Structure-Oriented Database Management System

1997-15 George Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms

1998-01 * Fachgruppe Informatik: Jahresbericht 1997

1998-02 Stefan Gruner, Manfred Nagel, Andy Schürr: Fine-grained and Structure-Oriented Document Integration Tools are Needed for Development Processes

1998-03 Stefan Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr

1998-04 * O. Kubitz: Mobile Robots in Dynamic Environments

1998-05 Martin Leucker, Stephan Tobies: Truth - A Verification Platform for Distributed Systems

1998-06 * Matthias Oliver Berger: DECT in the Factory of the Future
1998-07 M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects
1998-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien
1998-10 * M. Nicola, M. Jarke: Performance Modeling of Distributed and Replicated Databases
1998-11 * Ansgar Schleicher, Bernhard Westfechtel, Dirk Jäger: Modeling Dynamic Software Processes in UML
1998-12 * W. Appelt, M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web
1998-13 Klaus Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation
1999-01 * Jahresbericht 1998
1999-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version
1999-03 * R. Gallersdörfer, M. Jarke, M. Nicola: The ADR Replication Manager
1999-04 María Alpuente, Michael Hanus, Salvador Lucas, Germán Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing
1999-05 * W. Thomas (Ed.): DLT 99 - Developments in Language Theory Fourth International Conference
1999-06 * Kai Jakobs, Klaus-Dieter Kleefeld: Informationssysteme für die angewandte historische Geographie
1999-07 Thomas Wilke: CTL+ is exponentially more succinct than CTL
1999-08 Oliver Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures
2000-01 * Jahresbericht 1999
2000-02 Jens Vöge, Marcin Jurdzinski A Discrete Strategy Improvement Algorithm for Solving Parity Games
2000-03 D. Jäger, A. Schleicher, B. Westfechtel: UPGRADE: A Framework for Building Graph-Based Software Engineering Tools
2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach
2000-05 Mareike Schoop: Cooperative Document Management
2000-06 Mareike Schoop, Christoph Quix (eds.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling
2000-07 * Markus Mohnen, Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages
2000-08 Thomas Arts, Thomas Noll: Verifying Generic Erlang Client-Server Implementations
2001-01 * Jahresbericht 2000
2001-02 Benedikt Bollig, Martin Leucker: Deciding LTL over Mazurkiewicz Traces
2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04	Benedikt Bollig, Martin Leucker, Michael Weber: Local Parallel Model Checking for the Alternation Free mu-Calculus
2001-05	Benedikt Bollig, Martin Leucker, Thomas Noll: Regular MSC Languages
2001-06	Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic
2001-07	Martin Grohe, Stefan Wöhrle: An Existential Locality Theorem
2001-08	Mareike Schoop, James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling
2001-09	Thomas Arts, Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs
2001-10	Achim Blumensath: Axiomatising Tree-interpretable Structures
2001-11	Klaus Indermark, Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung
2002-01 *	Jahresbericht 2001
2002-02	Jürgen Giesl, Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems
2002-03	Benedikt Bollig, Martin Leucker, Thomas Noll: Generalised Regular MSC Languages
2002-04	Jürgen Giesl, Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting
2002-05	Horst Lichter, Thomas von der Maßen, Thomas Weiler: Modelling Requirements and Architectures for Software Product Lines
2002-06	Henry N. Adorna: 3-Party Message Complexity is Better than 2-Party Ones for Proving Lower Bounds on the Size of Minimal Nondeterministic Finite Automata
2002-07	Jörg Dahmen: Invariant Image Object Recognition using Gaussian Mixture Densities
2002-08	Markus Mohnen: An Open Framework for Data-Flow Analysis in Java
2002-09	Markus Mohnen: Interfaces with Default Implementations in Java
2002-10	Martin Leucker: Logics for Mazurkiewicz traces
2002-11	Jürgen Giesl, Hans Zantema: Liveness in Rewriting
2003-01 *	Jahresbericht 2002
2003-02	Jürgen Giesl, René Thiemann: Size-Change Termination for Term Rewriting
2003-03	Jürgen Giesl, Deepak Kapur: Deciding Inductive Validity of Equations
2003-04	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Improving Dependency Pairs
2003-05	Christof Löding, Philipp Rohde: Solving the Sabotage Game is PSPACE-hard
2003-06	Franz Josef Och: Statistical Machine Translation: From Single-Word Models to Alignment Templates
2003-07	Horst Lichter, Thomas von der Maßen, Alexander Nyßen, Thomas Weiler: Vergleich von Ansätzen zur Feature Modellierung bei der Softwareproduktlinienentwicklung
2003-08	Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, Stephan Falke: Mechanizing Dependency Pairs
2004-01 *	Fachgruppe Informatik: Jahresbericht 2003

2004-02 Benedikt Bollig, Martin Leucker: Message-Passing Automata are expressively equivalent to EMSO logic

2004-03 Delia Kesner, Femke van Raamsdonk, Joe Wells (eds.): HOR 2004 – 2nd International Workshop on Higher-Order Rewriting

2004-04 Slim Abdennadher, Christophe Ringeissen (eds.): RULE 04 – Fifth International Workshop on Rule-Based Programming

2004-05 Herbert Kuchen (ed.): WFLP 04 – 13th International Workshop on Functional and (Constraint) Logic Programming

2004-06 Sergio Antoy, Yoshihito Toyama (eds.): WRS 04 – 4th International Workshop on Reduction Strategies in Rewriting and Programming

2004-07 Michael Codish, Aart Middeldorp (eds.): WST 04 – 7th International Workshop on Termination

2004-08 Klaus Indermark, Thomas Noll: Algebraic Correctness Proofs for Compiling Recursive Function Definitions with Strictness Information

2004-09 Joachim Kneis, Daniel Mölle, Stefan Richter, Peter Rossmanith: Parameterized Power Domination Complexity

2004-10 Zinaida Benenson, Felix C. Gärtner, Dogan Kesdogan: Secure Multi-Party Computation with Security Modules

2005-01 * Fachgruppe Informatik: Jahresbericht 2004

2005-02 Maximillian Dornseif, Felix C. Gärtner, Thorsten Holz, Martin Mink: An Offensive Approach to Teaching Information Security: "Aachen Summer School Applied IT Security"

2005-03 Jürgen Giesl, René Thiemann, Peter Schneider-Kamp: Proving and Disproving Termination of Higher-Order Functions

2005-04 Daniel Mölle, Stefan Richter, Peter Rossmanith: A Faster Algorithm for the Steiner Tree Problem

2005-05 Fabien Pouget, Thorsten Holz: A Pointillist Approach for Comparing Honeypots

2005-06 Simon Fischer, Berthold Vöcking: Adaptive Routing with Stale Information

2005-07 Felix C. Freiling, Thorsten Holz, Georg Wicherski: Botnet Tracking: Exploring a Root-Cause Methodology to Prevent Distributed Denial-of-Service Attacks

2005-08 Joachim Kneis, Peter Rossmanith: A New Satisfiability Algorithm With Applications To Max-Cut

2005-09 Klaus Kursawe, Felix C. Freiling: Byzantine Fault Tolerance on General Hybrid Adversary Structures

2005-10 Benedikt Bollig: Automata and Logics for Message Sequence Charts

2005-11 Simon Fischer, Berthold Vöcking: A Counterexample to the Fully Mixed Nash Equilibrium Conjecture

2005-13 Carole Delporte-Gallet, Hugues Fauconnier, Felix C. Freiling: Revisiting Failure Detection and Consensus in Omission Failure Environments

2005-14 Felix C. Freiling, Sukumar Ghosh: Code Stabilization

2005-15 Uwe Naumann: The Complexity of Derivative Computation

2005-16 Uwe Naumann: Syntax-Directed Derivative Code (Part I: Tangent-Linear Code)

2005-17 Uwe Naumann: Syntax-directed Derivative Code (Part II: Intraprocedural Adjoint Code)

2005-18	Thomas von der Maßen, Klaus Müller, John MacGregor, Eva Geisberger, Jörg Dörr, Frank Houdek, Harbhajan Singh, Holger Wußmann, Hans-Veit Bacher, Barbara Paech: Einsatz von Features im Software-Entwicklungsprozess - Abschlußbericht des GI-Arbeitskreises "Features"
2005-19	Uwe Naumann, Andre Vehreschild: Tangent-Linear Code by Augmented LL-Parsers
2005-20	Felix C. Freiling, Martin Mink: Bericht über den Workshop zur Ausbildung im Bereich IT-Sicherheit Hochschulausbildung, berufliche Weiterbildung, Zertifizierung von Ausbildungsangeboten am 11. und 12. August 2005 in Köln organisiert von RWTH Aachen in Kooperation mit BITKOM, BSI, DLR und Gesellschaft fuer Informatik (GI) e.V.
2005-21	Thomas Noll, Stefan Rieger: Optimization of Straight-Line Code Revisited

* These reports are only available as a printed version.

Please contact `biblio@informatik.rwth-aachen.de` to obtain copies.