Low-Cost Error Containment and Recovery for Onboard Guarded Software Upgrading and Beyond

Ann T. Tai, *Member*, *IEEE*, Kam S. Tso, *Member*, *IEEE*, Leon Alkalai, *Member*, *IEEE*, Savio N. Chau, *Member*, *IEEE*, and William H. Sanders, *Fellow*, *IEEE*

Abstract—Message-driven confidence-driven (MDCD) error containment and recovery, a low-cost approach to mitigating the effect of software design faults in distributed embedded systems, is developed for onboard guarded software upgrading for deep-space missions. In this paper, we first describe and verify the MDCD algorithms in which we introduce the notion of "confidence-driven" to complement the "communication-induced" approach employed by a number of existing checkpointing protocols to achieve error containment and recovery efficiency. We then conduct a model-based analysis to show that the algorithms ensure low performance overhead. Finally, we discuss the advantages of the MDCD approach and its potential utility as a general-purpose, low-cost software fault tolerance technique for distributed embedded computing.

Index Terms—Guarded software upgrading, message-driven confidence-driven, global state consistency and recoverability, performance overhead, software fault tolerance, distributed embedded systems.

1 Introduction

Among the attributes of the new-generation embedded systems for future deep-space exploration, hardware reconfigurability and software upgradability are increasingly viewed as properties crucial to the survival of a spacecraft in an ultra-long-life mission. Those onboard-evolution related requirements in turn necessitate the availability of more flexible, sophisticated computer architectures for embedded avionics systems. Accordingly, the research and development efforts for evolvable avionics systems endeavor to explore scalable distributed embedded computing.

One of the major challenges that arise from onboard software upgrading is that of guarding the system against performance loss caused by interrupted services of the spacecraft or residual design faults introduced by an upgrade. The current practice of onboard software upgrade imposes severe unavailability on mission operation and provides no safeguard function. In the Mars Pathfinder mission, for example, the process for installing two minor patches in the binary code took two hours, during which the flight computer completely stopped functioning [1]. The costly performance loss and potential mission-failure risk associated with such blackout periods are unacceptable for NASA's future missions. The necessity of guarding software upgrade against performance loss is further exemplified by MCI WorldCom's 10-day frame relay outage in August 1999 [2]. The outage began four weeks after a scheduled upgrade to a new switching software that was intended to allow the network to handle increased traffic. The incident affected about 15 percent of MCI's network and 30 percent of its customers who rely on the high-speed frame relay.

To avoid the adverse impacts of unsuccessful upgrades, researchers have investigated dependable system upgrade methods for critical applications. For example, Sha et al. developed the Simplex architecture, which employed "analytic redundancy" to enable error recovery for upgraded software [3]. Powell et al. defined a Generic Upgradable Architecture for Real-Time Dependable Systems (GUARDS) [4]. This novel upgradable architecture allows a designer to choose redundancy degrees for system resources in various dimensions and to determine differing protection mechanisms for software components with different criticalities. Since the new-generation embedded systems for deep-space missions exploit distributed architectures and must meet severe power constraints, we aim to seek low performance cost, flexible methods for mitigating the effects of residual software design faults in a distributed computing environment. More specifically, our concern is the error contamination (caused by residual faults in an upgraded software component) among the processes that execute in an asynchronous fashion and cooperate through message passing. To the best of our knowledge, methods for error containment and recovery in distributed computing environments have not received adequate attention in prior work concerning dependable system upgrade (see [3], [4], for example) or dynamic program modification (see [5], for example).

- *A.T. Tai and K.S. Tso are with IA Tech, Inc., Los Angeles, CA 90024. E-mail: {a.t.tai, k.tso}@ieee.org.*
- *L. Alkalai and S.N. Chau are with the Jet Propulsion Laboratory, Pasadena, CA 91109. E-mail: {lalkalai, schau}@jpl.nasa.gov.*
- *W.H. Sanders is with the University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: whs@crhc.uiuc.edu.*

Manuscript received 15 Oct. 2000; revised 16 May 2001; accepted 19 June 2001.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 114409.

With the above motivation, we have developed a methodology called *guarded software upgrading* (*GSU*) [6]. To ensure low development cost, we take advantage of inherent system resource redundancies as the means of fault tolerance. Specifically, from the software perspective, we make use of an earlier version in which we have high confidence due to its long onboard execution time as a backup to protect the system when the new version enters mission operation; from the hardware perspective, we make use of the processor that otherwise would be idle during a noncritical mission phase during which onboard software upgrade takes place, allowing concurrent execution of the new and old versions of the application software component which is undergoing an upgrade. To mitigate the effect of residual faults in an upgraded software component that executes in a distributed computing environment, we take a crucial step in devising error containment and recovery methods by introducing the "confidence-driven" notion. This notion complements the message-driven (or "communication-induced") approach employed by a number of existing checkpointing protocols for tolerating hardware faults [7], [8]. The resulting error containment and recovery protocol is thus both message-driven and confidence-driven (MDCD). In particular, the MDCD protocol is based on a two-tiered approach: First, we discriminate among software components with respect to our confidence in them and, second, during onboard execution time, we adjust our confidence in the processes that are created from those software components, according to our knowledge about potential process state contamination caused by errors in a low-confidence component and message passing. The latter implies that we judge the "trustworthiness" of a process in a dynamic fashion.

Error containment algorithms for mitigating the effect of design faults in a distributed computing environment must ensure that the system will reach a *valid* consistent global state in case of error recovery. This criterion often leads a software fault tolerance scheme to sacrifice much of the performance advantage of distributed computing, due to the costs derived from process coordination and synchronization. The MDCD approach allows interacting processes to communicate with each other without restriction while keeping track of potential error contamination and ensuring that the most recent noncontaminated state of a process is always kept available to enable correct error recovery actions. During error recovery, processes are allowed to decide whether to roll back or roll forward based on the locally available knowledge about potential process state contamination, without having to consult a message-exchange based global decision algorithm for state restoration. In [9], we conducted a reliability analysis that demonstrated the effectiveness of the MDCD protocol with respect to enhancing reliability of the embedded system when it undergoes onboard software upgrading. The model-based evaluation presented in this paper shows that dynamic confidence adjustment plays a crucial role in performance overhead reduction.

Indeed, there are a number of factors other than upgrading, such as complexity or testability, that may lead us to discriminate among interacting software components in a distributed system with respect to our confidence in their trustworthiness. Also, some software component in a distributed system may have a higher message-sending rate than the others, which implies that an error of that component is more likely to propagate; such a component could become the critical component of the system in the sense that it dominates error contamination and should thus be given priority for fault tolerance. Those factors, combined with the MDCD protocol's ability to facilitate the application of software fault tolerance to selected interacting components, suggest that the MDCD approach can be utilized as a general-purpose low-cost software fault tolerance technique for distributed computing.

The remainder of the paper is organized as follows: Section 2 provides an overview of the GSU framework. Sections 3 and 4 present the algorithm description and verification, respectively, followed by Section 5, which illustrates a model-based performance cost analysis. Section 6 compares our work with the prior efforts in related areas. The concluding remarks summarize what we have accomplished and describe the issues which we are currently investigating.

2 Background: GSU Framework

The GSU framework is motivated by the Baseline X2000 First Delivery Architecture [10], which is comprised of three high-performance computing nodes (each of which has a local DRAM) and multiple subsystem microcontroller nodes interfacing with a variety of devices. All nodes are connected by a fault-tolerant bus network that complies with the commercial interface standard IEEE 1394, facilitating reliable onboard distributed computing.

Since a scheduled software upgrade is normally conducted during a cruise phase when the spacecraft and science functions do not require full computation power, only two processes, corresponding to two functionally different application software components, will run concurrently and interact with each other during this phase. To exploit inherent system resource redundancies, we let the old version (of the component that is undergoing an upgrade), in which we have high confidence due to its long onboard execution time, escort the new version through *onboard validation* and *guarded operation*. Further, we make use of the processor that otherwise would be idle to enable the three processes (i.e., the two corresponding to the new and old versions and the process corresponding to the second application software component) to execute concurrently. To aid in the description, we introduce the following notation:

- P_1^{new} The process corresponding to the new version of an application software component.
- P_1^{old} The process corresponding to the old version of the application software component.
- P_2 The process corresponding to another application software component (which is not undergoing upgrade).

Fig. 1 illustrates the two-stage approach. In order to minimize the impact and risk on mission operation, onboard software upgrading is usually carried out in an

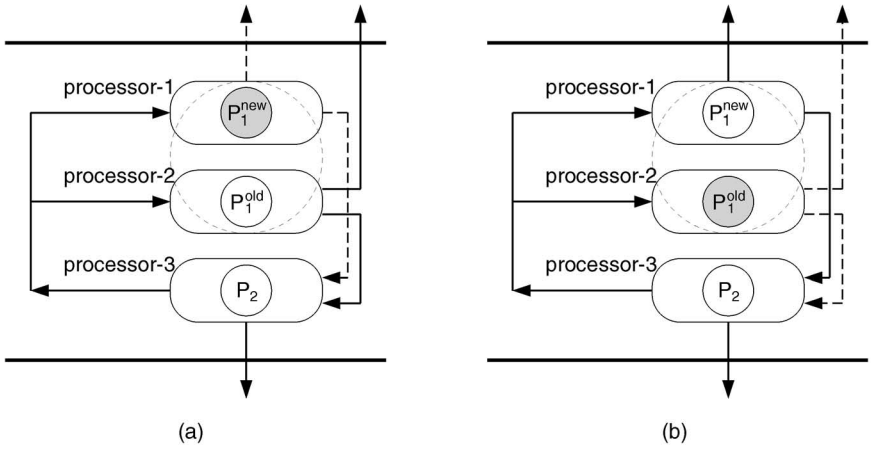


Fig. 1. Two-stage approach to GSU. (a) Onboard validation stage. (b) Guarded operation stage.

incremental fashion. In particular, onboard upgrades for spaceborne systems typically involve only a single software component at a time. As a result, the interaction patterns (message types and ordering) among the processes will remain the same after an upgrade. Accordingly, as shown in Fig. 1a, during onboard validation, the outgoing messages of the shadow process P_1^{new} are suppressed but selectively logged (as shown by the dashed lines with arrows), while P_1^{new} receives the same incoming messages that the active process P_1^{old} does (as shown by the solid lines with arrows). Thus, P_1^{new} and P_1^{old} can perform the same computation based on identical input data. Note that the dashed circles that encapsulate P_1^{new} and P_1^{old} indicate that the two processes are created by two different versions of the same application software component.

By maintaining an onboard error log that can be downloaded to the ground to facilitate statistical modeling and heuristic trend analysis, onboard validation facilitates the decisions on whether and when to permit P_1^{new} to enter mission operation. If onboard validation completes successfully, then P_1^{new} and P_1^{old} switch their roles to enter the guarded operation stage.

During guarded operation, P_1^{new} starts to actually influence the external world and interact with process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or devices are suppressed and logged, as indicated in Fig. 1b. Should an error of P_1^{new} be detected, P_1^{old} will take over P_1^{new}'s active role and the system will resume its normal mode. The guarded operation is enabled by an error containment and recovery protocol that is described in the next section.

3 MDCD Error Containment and Recovery Protocol

3.1 Assumptions

A major difficulty in error recovery for embedded systems is that we are unable to roll back the effect of a computation error after it propagates to an external device. Since error propagation in a distributed system is, in general, caused by message passing, the invocations of the two major functions of the error containment and recovery protocol for GSU, namely, checkpointing and acceptance test (AT), are all associated with the message sending or receiving actions. We call the messages sent by processes to devices and the messages between processes *external messages* and *internal messages*, respectively. In embedded systems, external messages are significantly more critical than internal messages because 1) they directly influence the mission operation and functions and 2) their adverse effects cannot be reversed through rollback. Hence, in our low-cost error containment and recovery protocol, ATs are only invoked to validate the external messages from the processes that are potentially contaminated (see Section 3.2 for the definition of *potentially contaminated process state*). Further, P_1^{old} does not perform ATs because its external messages will not be released to devices during guarded operation. On the other hand, when P_1^{new} or P_2 passes an AT successfully, it sends a notification message to P_1^{old} to let it update its knowledge about the validity of process state and messages.

The following are the assumptions upon which we devise the error containment and recovery algorithms:

A1. The old version of a software component that has a sufficiently long onboard execution time can be considered significantly more reliable than the upgraded version newly installed through uploading.
A2. An erroneous state of a process is likely to affect the correctness of its outgoing messages, while an erroneous message received by an application software component will result in process state contamination.
A3. The error detection mechanism, acceptance test, has a high coverage (the conditional probability that the testing mechanism will reject a computation result given that the result is erroneous).

A1 implies that the likelihood that an error condition will occur in the old version of a software component can be considered negligible, suggesting that P_1^{old} and P_2 need not be treated by the protocol as possible sources of process state contamination. A2 implies that if an outgoing message is validated by AT, then the process state of the sending process and all the messages sent or received by the process prior to the AT-based validation can be considered *noncontaminated* and *valid*, respectively. A3 suggests that the release of an erroneous command to an external device

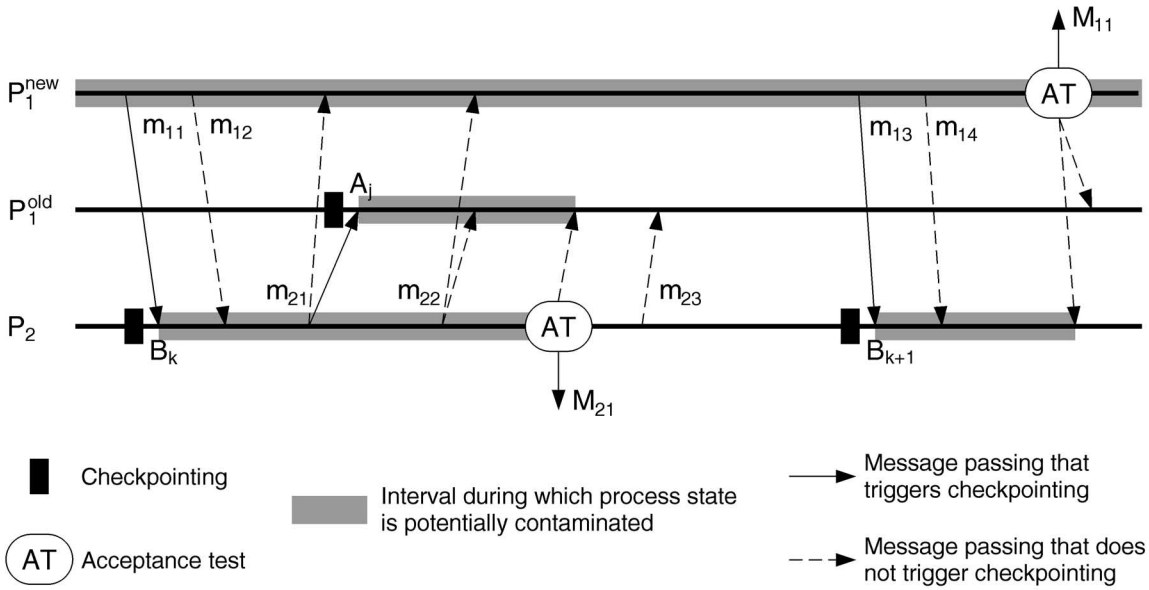


Fig. 2. MDCD approach to error containment.

is unlikely to occur. Note that A1 is applicable not only to the upgrades for performance tuning and accuracy improvement but also to the *scheduled upgrades* aimed at fault removal [11]. The rationale is that the deep-space application software components which have sufficiently long onboard execution times are expected to be highly reliable and that the scheduled onboard fault removal usually deals with the isolated faults that result in infrequent error conditions tolerable by the spaceborne system. On the other hand, the new version with a known fault removed may contain new undiscovered faults.

3.2 Error Containment Algorithms

3.2.1 Concepts

During guarded operation, P_1^{new} actually influences the external world and interacts with the process P_2, while the messages of P_1^{old} that convey its computation results to P_2 or external subsystems are suppressed and logged (although P_1^{old} receives all the messages that P_1^{new} does and fully executes in the background). The algorithms rely on three key entities, namely,

1. a *dirty bit* (`dirty_bit`), which keeps track of the dynamically adjusted confidence in a process and represents the knowledge about potential process state contamination,
2. *message sequence numbers* ($msg\_SN\_P_1^{new}$ and $msg\_SN\_P_1^{old}$), which keep track of messages sent by the new and old versions of the software component that is undergoing an upgrade,
3. a *valid message register* (VR_1^{new}), which keeps track of the sequence number of the last message that has been sent by the low-confidence process P_1^{new} and directly or indirectly validated by an AT.

These entities are maintained by individual processes locally while the information kept is shared between certain processes through message piggybacking. To facilitate error containment and recovery efficiency, we enforce the following confidence-driven checkpointing rule and AT-based validation policy:

Checkpointing Rule: A process must save its state in a checkpoint if and only if the process is at the following point: immediately before its otherwise noncontaminated state becomes potentially contaminated.

AT-Based Validation Policy: We use an AT to validate a message that a process intends to send if and only if: 1) The message is an external message and 2) the state of the sending process is potentially contaminated when the message is generated.

The definition of a *potentially contaminated process state* is as follows:

Definition 1. *By a "potentially contaminated process state," we mean 1) the process state of P_1^{new} in which we have not yet established enough confidence or 2) a process state that reflects the receipt of a not-yet-validated message that is sent by a process when its process state is potentially contaminated.*

Correspondingly, we use the term "a noncontaminated process state" to refer to a process state that is not potentially contaminated.

The above definition implies that the process created from the low-confidence software component, P_1^{new}, will invariably be considered potentially contaminated during guarded operation, while our confidence in a process created from a high-confidence component can be altered by a message passing event. In the interest of bridging terminology, the terms "a potentially contaminated process" and "a low-confidence process" are regarded as interchangeable in the remainder of this paper. Likewise, when we say "a high-confidence process," we mean a process that has a noncontaminated state. Accordingly, a low-confidence software component always corresponds to a low-confidence process, but the reverse is not true.

Fig. 2 illustrates the above concepts and defines the notation convention. In the figure, an upper case letter is

used to denote checkpoints of a particular process, while subscripts are used to represent the ordering of the checkpoints (of the same process). Thus, if B_i and B_j denote two checkpoints of P_2 established at times t' and t'', respectively, then $i < j$ implies $t' < t''$, while m_{ij} and M_{ij} denote, respectively, the jth application-purpose internal and external messages sent by process P_i. If m_{ij} and m_{ik} (or M_{ij} and M_{ik}) represent two internal (or external) messages sent by P_i at times t' and t'', respectively, then $j < k$ implies $t' < t''$.

The horizontal lines in the diagram represent the software executions along the time horizon. Each of the shaded regions represents an execution interval during which the state of the corresponding process is potentially contaminated. Note that 1) P_1^{new} is always considered potentially contaminated throughout guarded operation, 2) P_2 is treated as potentially contaminated after it receives an application-purpose message from P_1^{new} and before P_2 passes the subsequent AT (or receives a notification message reporting that P_1^{new} passes AT), and 3) P_1^{old} is regarded as potentially contaminated after it receives a message sent by P_2 when its process state is potentially contaminated and before P_1^{old} is notified that P_2 (or P_1^{new}) has passed the subsequent AT.

In the example scenario illustrated in Fig. 2, checkpoints B_k, A_j, and B_{k+1} are established immediately before a process state becomes potentially contaminated. Note that those checkpoint establishments are triggered by message passing events which change our confidence in a process. In other words, a message passing event will not trigger a process to establish a checkpoint unless the event alters our confidence in a process state. Therefore, our algorithms are both message-driven and confidence-driven. Consequently, as shown in Fig. 2, the message passing events indicated by the dashed lines with arrows will not trigger checkpointing. And, based on our assumption on the relationship between message and process state, m_{11}, m_{12}, m_{21}, and m_{22} become *valid messages* after P_2 subsequently passes AT for M_{21}. Likewise, m_{13} and m_{14} are regarded as valid messages after P_1^{new} subsequently passes AT for M_{11}. On the other hand, the message m_{23} sent by P_2 is considered an "inherently" valid message because it is generated by P_2 when the process state of P_2 is not potentially contaminated. Without losing generality, P_1^{new} is exempted from performing checkpointing because we invariably view it as a low-confidence process throughout guarded operation.

The original version of the MDCD protocol [12] also requires a process to establish a checkpoint right after its potentially contaminated state is validated by an AT as a noncontaminated state. Those checkpoints are called "Type-2" checkpoints and are intended to be saved to stable storage for tolerating hardware faults. Recently, we have developed a scheme that enables synergistic coordination between the MDCD protocol and a time-based checkpointing protocol for simultaneous tolerance of software and hardware faults. The updated version of the MDCD protocol presented here thus involves no Type-2 checkpoints.

Since embedded applications are often associated with physical laws and usually exhibit mathematical properties (e.g., strong relations over inputs, outputs, and states) and produce computation results that have more restrictive ranges and states relative to nonembedded applications, the design of effective and efficient ATs for embedded systems is, in general, considerably easier than that for nonembedded systems [13], [14], [15]. Therefore, limiting the use of AT to external messages plays a dual role in performance cost reduction. Specifically, this strategy not only enables a process to perform AT less frequently, but also facilitates testing effectiveness and efficiency. This is because external messages are the ultimate computation results of an embedded application, unlike internal messages, which convey intermediate computation results that are far more difficult to validate.

3.2.2 Algorithm Description

The error containment algorithms for P_1^{new}, P_1^{old}, and P_2 are shown in Figs. 3, 4, and 5, respectively. Recall that our key strategy for achieving error containment and recovery efficiency is to discriminate between the individual software components with respect to our confidence in them. Accordingly, with the algorithms, the three interacting processes behave in an asymmetric fashion, as described in the following.

As shown in Fig. 3, P_1^{new} maintains its message sequence number $\texttt{msg\_SN\_P}_1^{new}$, which keeps track of the outgoing messages of P_1^{new} and is incremented when P_1^{new} sends an application-purpose message. To ensure consistency and recoverability, P_1^{new} attaches the value of $\texttt{msg\_SN\_P}_1^{new}$ to its outgoing messages sent to other processes. Note that the `dirty_bit` of P_1^{new} has a constant value of 1, indicating that the process is invariably considered a low-confidence process until guarded operation completes successfully. In accordance with the confidence-driven checkpointing rule and AT-based validation policy, P_1^{new} never performs checkpointing, whereas it conducts AT for every external message it intends to send.

P_1^{old} maintains message sequence numbers $\texttt{msg\_SN\_P}_1^{old}$ and $\texttt{msg\_SN\_P}_1^{new}$ (see Fig. 4). The variable $\texttt{msg\_SN\_P}_1^{old}$ keeps track of P_1^{old}'s own outgoing messages and is incremented after P_1^{old} generates an outgoing message and before the message is logged; the update of $\texttt{msg\_SN\_P}_1^{new}$ is carried out according to the piggybacked information, when P_1^{old} receives a message. In addition, P_1^{old} has a valid message register $\texttt{VR}_1^{new}$, which keeps track of validated messages sent by P_1^{new}. This register is updated when P_1^{old} receives a "passed AT" notification message from P_1^{new} or P_2, based on the piggybacked $\texttt{msg\_SN\_P}_1^{new}$. If P_1^{old} receives a "passed AT" message (from P_1^{new} or P_2) that piggybacks a $\texttt{msg\_SN\_P}_1^{new}$ greater than or equal to P_1^{old}'s local $\texttt{msg\_SN\_P}_1^{new}$, P_1^{old} deletes its checkpoint[1] and logged messages and then resets its `dirty_bit`. Conversely, upon the arrival of a message from P_2 with the piggybacked `dirty_bit` indicating that the sending process is potentially contaminated, P_1^{old} will, if its `dirty_bit` equals zero and its local $\texttt{msg\_SN\_P}_1^{new}$ is less than that piggybacked on the incoming message, save its

1. The checkpoint deletion is indeed optional because, with the algorithms, a process will roll back no further than its most recent checkpoint and, thus, we can always let a checkpoint be overwritten by the subsequent checkpoint.

```
// P1^new's dirty bit has a constant value of 1
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (AT(m) == success) {
      msg_SN_P1^new++;
      msg_sending(m, null, null, device);
      msg_sending("passed_AT", null, msg_SN_P1^new, {P1^old, P2}); // prior messages are valid
    } else {
      error_recovery(P1^old, P2);
      exit(error);
    }
  } else { // m is an internal message
      msg_SN_P1^new++;
      msg_sending(m, dirty_bit, msg_SN_P1^new, P2);
  }
}
if (incoming_message_queue_nonempty) {
  application_msg_reception(m);
}
```

Fig. 3. Error containment algorithm for $\mathrm{P}_1^{\mathrm{new}}$.

state via checkpointing and then set its dirty_bit to 1 before passing the message to the application. Note that, to compare the piggybacked msg_SN_$\mathrm{P}_1^{\mathrm{new}}$ with the local msg_SN_$\mathrm{P}_1^{\mathrm{new}}$ before $\mathrm{P}_1^{\mathrm{old}}$ updates its dirty_bit is necessary. This is because, due to the asynchronous nature of message passing and message-delivery delay, messages from different senders (e.g., an application-purpose message from P_2 and a "passed AT" notification message from $\mathrm{P}_1^{\mathrm{new}}$) may arrive at the receiver site in a nondeterministic order and cause incorrect confidence adjustment. Nonetheless, for guaranteeing predictable latency, the typical bus interfaces used by the embedded systems in real-time control environments will not allow alternative routes for a given sender-receiver pair during normal operation and will not apply store-and-forward packet transmission. As a result, messages between a particular sender and a particular receiver can never be received out-of-order. Hence, P_2 and $\mathrm{P}_1^{\mathrm{new}}$ do not need the similar sequence number check because P_2 receives messages only from $\mathrm{P}_1^{\mathrm{new}}$ and vice versa.

P_2 also keeps track of outgoing messages of $\mathrm{P}_1^{\mathrm{new}}$ by maintaining a local variable msg_SN_$\mathrm{P}_1^{\mathrm{new}}$, as shown in Fig. 5.

```
if (outgoing_message_m_ready) {
  msg_SN_P1^old++;  // msg_SN_P1^old keeps track of P1^old's own messages
  msg_logging(m, msg_SN_P1^old, msg_log); // suppress and log the outgoing message
}
if (incoming_message_queue_nonempty) {
  if (m.body == "passed_AT") {
    if (m.msg_SN_P1^new ≥ msg_SN_P1^new) {
      memory_reclamation(m.msg_SN_P1^new, ckpt, msg_log);
      dirty_bit = 0;
    }
    VR1^new = max(m.msg_SN_P1^new, VR1^new); // last valid message of P1^new
  } else { // application-purpose message from P2
    if (m.dirty_bit == 1 && dirty_bit == 0 && m.msg_SN_P1^new > msg_SN_P1^new) {
      checkpointing(P1^old);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
  msg_SN_P1^new = max(m.msg_SN_P1^new, msg_SN_P1^new);
}
```

Fig. 4. Error containment algorithm for $\mathrm{P}_1^{\mathrm{old}}$.

```
if (outgoing_message_m_ready) {
  if (external(m)) {
    if (dirty_bit == 1) {
      if (AT(m) == success) {
        dirty_bit = 0;
        memory_reclamation(msg_SN_P1^new, ckpt, null);
        msg_sending(m, null, null, device);
        msg_sending("passed_AT", null, msg_SN_P1^new, P1^old);
      } else {
        error_recovery(P1^old, P2);
      }
    } else { // outgoing msg from a clean state, no check needed
      msg_sending(m, null, null, device);
    }
  } else { // piggyback dirty_bit on internal msg to signal potential contamination
    msg_sending(m, dirty_bit, msg_SN_P1^new, {P1^new, P1^old});
  }
}
if (incoming_message_queue_nonempty) { // must be from P1^new
  if (m.body == "passed_AT") {
    if (dirty_bit == 1) {
      dirty_bit = 0;
      memory_reclamation(m.msg_SN_P1^new, ckpt, null);
    }
  } else {
    if (dirty_bit == 0) {
      checkpointing(P2);
      dirty_bit = 1;
    }
    application_msg_reception(m);
  }
  msg_SN_P1^new = m.msg_SN_P1^new;
}
```

Fig. 5. Error containment algorithm for P_2.

Upon receiving an application-purpose message or a "passed AT" notification message from P_1^{new}, msg_SN_P_1^{new} is updated according to the piggybacked information. Upon passing an AT, P_2 attaches the value of msg_SN_P_1^{new} to the "passed AT" notification message to P_1^{old} such that P_1^{old} can update VR_1^{new} accordingly. When P_2 passes an AT or receives a "passed AT" message from P_1^{new}, P_2 resets its dirty_bit and deletes its previously established checkpoint if P_2 is potentially contaminated prior to the AT; whereas if an application-purpose message from P_1^{new} arrives when P_2 has a noncontaminated state, P_2 establishes a checkpoint and then sets its dirty_bit to 1 before passing the message to the application. The value of dirty_bit is piggybacked to every application-purpose message that P_2 sends to P_1^{old} (and P_1^{new}) in order to let P_1^{old} be aware of potential process state contamination and be able to make a correct decision on whether to roll back or roll forward when error recovery is invoked.

3.3 Error Recovery Algorithms

Error recovery actions are also message-driven and confidence-driven, in the sense that an AT-based validation takes place when a potentially contaminated process (P_1^{new} or P_2) attempts to send an external message. If an error is detected by AT, P_1^{old} will take over P_1^{new}'s active role and resume computation with P_2 after error recovery. Since the error containment algorithms enable the processes to maintain their knowledge about potential process state contamination and message validity, the error recovery algorithms for P_1^{old} and P_2 can be very simple, as shown in Figs. 6 and 7, respectively. In particular, by checking their own dirty bits, both P_1^{old} and P_2 are able to make their decisions on rollback or roll-forward locally in a straightforward manner. There are three possible outcomes:

Scenario 1: Both P_1^{old} and P_2 roll forward.

Scenario 2: P_2 rolls back to its most recent checkpoint, while P_1^{old} rolls forward.

Scenario 3: Both P_1^{old} and P_2 roll back to their most recent checkpoints.

By "roll forward," we mean that a process continues its execution from its current state when error recovery is invoked. Note that the scenario in which P_1^{old} rolls back and

```
if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// take over from P1^new and go forward
switch_to_active(VR1^new, msg_SN_P1^old);
continue;
```

Fig. 6. Error recovery algorithm for P_1^{old}.

P_2 rolls forward can never happen, as shown by the proof for Lemma 1 in Section 4.

After the rollback action or the roll-forward decision, P_1^{old} will compare the values of $msg\_SN\_P_1^{old}$ and VR_1^{new}. Depending upon whether the former is greater or less than the latter, P_1^{old} will either send out the messages that are saved in the message log and have sequence numbers greater than the value of VR_1^{new}, or continue to suppress its messages, respectively, until the values of $msg\_SN\_P_1^{old}$ and VR_1^{new} match.

4 Algorithm Verification

Clearly, our error containment and recovery algorithms are based on the adaptation and integration of a number of existing fault tolerance techniques. For example, we use 1) multiple software versions (which are inherently available to us) and 2) acceptance tests, as suggested by N-version programming (NVP) [16] and recovery blocks (RB) [17], respectively. However, rather than being driven by the mechanisms prestructured in an application program as suggested by traditional software fault tolerance schemes, checkpointing and error recovery in our algorithms are triggered by message passing events, which is a strategy adapted from the checkpointing techniques for hardware fault tolerance [7]. Nonetheless, checkpointing techniques for hardware error recovery concern solely the errors resulting from hardware faults and the consistency between process states themselves, rather than design-fault-caused errors in process states. In contrast, since our objective is to mitigate the effect of residual design faults in an upgraded software component, our particular concern is the consistency among the views of different processes on process state integrity, especially on *validity* of the messages (see Sections 3.1 and 3.2.1) reflected in the process states. Accordingly, the notion of confidence-driven is the key to our adaptation of enabling techniques.

Specifically, we adapt the terminologies and definitions in [7], [18] as follows: A *global state* includes the state of each process that is executing the application and, possibly, messages between interacting processes and *information concerning their verified correctness*. A valid checkpointing mechanism must ensure that it is always possible for the error recovery mechanism to bring the system into a global state that satisfies the following two properties:

Consistency: If, in a global state S, m is reflected as a message received by a process, then m must also be reflected in S as a message sent by the sending process, and the sending and receiving processes must have consistent views on the validity of m.

```
if (dirty_bit == 1) {
  rollback(most_recent_ckpt);
}
// go forward
continue;
```

Fig. 7. Error recovery algorithm for P_2.

Recoverability: If, in a global state S, m is reflected as a message sent by a process, then m must also be reflected in S as a message received by the receiving process(es), and the sending and receiving processes must have consistent views on the validity of m, or the error recovery algorithm must be able to restore m.

When two or more process states (or checkpoints reflecting the process states) comprise a global state that satisfies the consistency property, we say that these process states are *globally consistent* or that they comprise a *consistent global state*. It is worth noting that, upon error recovery, P_1^{old} takes over P_1^{new}'s active role and thus becomes the "sending process" of the messages sent by P_1^{new} and reflected as valid messages in the global state.

The theorems and proofs presented below show formally that the rollback or roll-forward recovery decisions made locally by the individual processes guarantee the global state consistency and recoverability properties. As the error containment algorithms require a potentially contaminated process to perform AT for its outgoing external messages and to keep other processes informed of successful external message sending, the information regarding external message passing reflected in a global state will never violate global state consistency. Therefore, the proofs presented below are concerned solely with internal messages. Further, unless explicitly stated, all the scenarios referred to in the theorems and proofs are those that occur prior to error recovery.

Theorem 1. *The process states of P_1^{old} and P_2 at time t that are not potentially contaminated are globally consistent.*

Proof. From the definition of potentially contaminated process state, a noncontaminated process state will not reflect any not-yet-validated messages that are sent by the process when it is potentially contaminated or that are received from a potentially contaminated process. This implies that, based on their noncontaminated process states, P_1^{old} and P_2 will have consistent views on message validity. Then, the theorem follows directly from the definition of global state consistency. □

Lemma 1. *The scenario in which the process state of P_1^{old} is potentially contaminated but that of P_2 is not will not occur.*

Proof. Since P_1^{new} and P_1^{old} are created from two functionally similar application software components, there will not be any application-purpose interprocess communication between them. This in turn implies that the process state of P_1^{old} will not become potentially contaminated unless it receives a message sent by P_2 when the process state of P_2 is potentially contaminated. □

Theorem 2. *If, at time t, the process state of P_2 is potentially contaminated but that of P_1^{old} is not, then the process state of P_1^{old} at time t and the process state of P_2 reflected in its most recent checkpoint (relative to t) are globally consistent.*

Proof. If, at time t, the process state of P_2 is potentially contaminated, then the most recent checkpoint (relative to t) of P_2, call it B_k, must be established immediately before P_2 becomes potentially contaminated. As implied by Lemma 1 and Theorem 1, the process states of P_2 and P_1^{old} immediately before P_2 establishes B_k must both be noncontaminated and thus are globally consistent. If a message m is reflected in the process state of P_1^{old} at time t as a valid message received from P_2, m must also be reflected in B_k as a valid outgoing message (sent to P_1^{old} and P_1^{new}). This is because m must be sent by P_2 before it establishes B_k; otherwise, the process state of P_1^{old} at time t would have been potentially contaminated, which is a contradiction.

Conversely, if B_k reflects a valid message m' that is received by P_2 from P_1^{new}, then m' must be reflected (through the value of $\texttt{VR}_1^{new}$) in the process state of P_1^{old} at time t as a valid message sent by P_1^{new}. This is because m' must have been validated by an AT (performed by P_2 or P_1^{new}) and, upon this successful AT, P_1^{old} must receive a "passed AT" notification message and update its $\texttt{VR}_1^{new}$ according to the piggybacked sequence number of the last validated message of P_1^{new}. □

Corollary 1. *If the process states of P_1^{old} and P_2 at time t are potentially contaminated, the process states reflected in their most recent checkpoints (relative to t) are globally consistent.*

Proof. Let the most recent checkpoints of P_1^{old} and P_2 (relative to t) be denoted by A_j and B_k, respectively. If P_1^{old} and P_2 are potentially contaminated at time t, then A_j and B_k must be established immediately before P_1^{old} and P_2 become potentially contaminated, respectively. Suppose that A_j and B_k are established at times t'' and t', respectively. Then, per Lemma 1, it must be the case that $t'' > t'$. By Theorem 2, the process state of P_1^{old} at t'' and that of P_2 reflected in B_k are globally consistent. This, in turn, implies that the process states reflected in A_j and B_k are globally consistent. Hence, the corollary. □

As to recoverability, it is ensured by the entities $\texttt{msg\_SN\_P}_1^{old}$, $\texttt{VR}_1^{new}$, and `msg_log` (the message log where P_1^{old} saves its suppressed messages). Specifically, the value of $\texttt{VR}_1^{new}$ identifies the valid messages which are sent by P_1^{new} and "permanently received" by P_2 or an external subsystem (in other words, they will never be "unreceived" through rollback recovery). On the other hand, $\texttt{msg\_SN\_P}_1^{old}$ may be decremented or remain the same after recovery, depending upon whether P_1^{old} rolls back or forward, respectively. As explained earlier, P_1^{old} will "resend" the messages in its message log or suppress the messages it intends to send after recovery, if the value of $\texttt{msg\_SN\_P}_1^{old}$ is greater or less than that of $\texttt{VR}_1^{new}$, respectively, until the two values match, guaranteeing recoverability. The formal assertion and reasoning are given below.

Theorem 3. *The MDCD protocol guarantees that, upon the completion of the rollback or roll-forward actions carried out by individual processes, the system will reach a global state that satisfies the recoverability property.*

Proof. The type of event that will violate the recoverability property is the following: Upon the completion of the rollback or roll-forward error recovery actions carried out by individual processes, a message m is reflected in the sender's process state as a valid message already sent, but the receiving process has rolled back to a checkpoint that is established prior to the receipt of m. Thus, it will be sufficient for the proof if we can show that the error recovery scenarios that involve process rollback, namely, Scenarios 2 and 3 described in Section 3.3, will not lead to any such events.

In Scenario 2, P_2 rolls back to its most recent checkpoint, call it B_k, while P_1^{old} rolls forward. Then, any prior messages from P_1^{new} with sequence numbers greater than the value of $\texttt{msg\_SN\_P}_1^{new}$ reflected in B_k will be "unreceived" by P_2 due to its rollback. Suppose that error recovery is invoked at time t and B_k is established at time t' (immediately before P_2 becomes potentially contaminated) such that $t' < t$. Then, according to the error containment algorithms, the value of $\texttt{VR}_1^{new}$ reflected in the noncontaminated state of P_1^{old} at time t and the value of $\texttt{msg\_SN\_P}_1^{new}$ reflected in B_k to which P_2 rolls back must both be based on the last successful AT performed (by P_1^{new} or P_2) prior to t', implying that these two values are equal. Since P_1^{old} will resend the suppressed messages that are saved in its message log and have sequence numbers greater than the value of $\texttt{VR}_1^{new}$, all the "unreceived" messages caused by the rollback of P_2 will be restored.

In Scenario 3, P_1^{old} and P_2 roll back to their most recent checkpoints, call them A_j and B_k, respectively. A_j and B_k must be established immediately before P_1^{old} and P_2 become potentially contaminated, respectively. By Lemma 1, if the establishment of B_k is triggered by a message m' from P_1^{new} at time t', A_j must be triggered by a message m'' from P_2 at time t'' such that $t'' > t'$. Then, according to the checkpointing rule, m' must be the first message P_1^{new} sends to P_2 since the last successful AT performed in the system. From the error containment algorithms, it follows that both the value of $\texttt{VR}_1^{new}$ reflected in A_j and the value of $\texttt{msg\_SN\_P}_1^{new}$ reflected in B_k must be based on the last successful AT and, thus, the values are equal. Hence, the circumstance in which a message m is reflected in A_j as a valid message sent by P_1^{new}, but m is not reflected in B_k as a valid message received from P_1^{new} will never happen.

The theorem then follows from the definition of recoverability. □

The above theorems and proofs imply that the MDCD protocol guarantees that the system will reach a global state that satisfies the consistency and recoverability properties upon the completion of the rollback or roll-forward actions carried out by individual processes during error recovery. It is worth noting that the global state consistency and recoverability can further guarantee that the system will be failure-free if the MDCD protocol is run in an ideal execution environment. By an "ideal

execution environment," we mean an execution environment for the protocol that satisfies the following criteria: C1) P_1^{old} and P_2 are perfectly reliable, C2) error conditions in a process state will be definitely manifested in the messages sent by the corresponding process, and C3) each AT has a perfect coverage.

Clearly, criteria C1, C2, and C3 for the ideal execution environment of the MDCD protocol are a stronger version of Assumptions A1, A2, and A3, upon which we have based the design of the protocol (see Section 3.1). The implication is that violation of those criteria may have an effect on the protocol's fault tolerance capability. Specifically, an imperfect coverage of AT (which violates C3) may cause an erroneous external message to go undetected. And, a fault in P_1^{old} or P_2 (which is not allowed by C1) may result in an erroneous external message that is exempted from undergoing an AT-based validation. Also, if error manifestation in messages is indeterministic (contrary to C2), we may encounter dormant error conditions that lead to false confidence in process states. As the realistic goal of the MDCD protocol is to significantly reduce the probability of system failure rather than to guarantee the system to be failure-free, the protocol is anticipated to be effective in a nonideal execution environment. In order to validate the effectiveness of the protocol when it is run in an environment where C1, C2, and C3 are not satisfied, we conducted a model-based reliability analysis in [9]. The evaluation results confirm that the MDCD approach is effective with respect to reliability enhancement as originally surmised, even in a nonideal execution environment.

5 Analysis of Performance Cost

The objective of the evaluation presented below is to analyze the performance overhead reduction resulting from use of dynamic confidence adjustment. Accordingly, we choose to compare the MDCD protocol with an MDCD variant that employs a static confidence-driven approach. Due to the dynamic nature of the MDCD protocol, the evaluation requires a model to capture numerous interdependencies among system attributes. Therefore, as we did with the reliability analysis for the MDCD protocol [9], we choose to use stochastic activity networks (SANs) [19] to conduct the performance cost analysis because of their capability of explicitly representing dependencies among system attributes.

5.1 SAN Model for Evaluating Performance Overhead

Although SANs' rich syntax and marking-dependent specification capability allow us to specify every aspect of the protocol precisely, the resulting state space may become unmanageable if we attempt to make the SAN model a procedural specification of the MDCD protocol. To avoid the problem, our approach is to minimize explicit representation of the algorithmic details while ensuring that every aspect of their impact on the particular measure we seek to solve is captured. In contrast with the reliability analysis conducted in [9], the objective of this model is to evaluate the performance overhead during failure-free guarded operation. Accordingly, we omit those failure-behavior-related aspects, such as fault manifestation, undetected error, and dormant error conditions. Instead, we focus on representing those conditions that would lead a process to take error containment actions that contribute to performance overhead. This "measure-adaptive" approach to model construction prevents the state space from becoming unnecessarily large.

Fig. 8 depicts the SAN model for the evaluation of performance overhead. The model includes the timed and instantaneous activities that represent the error containment actions of the protocol that are driven by the message passing events and dynamically adjusted confidence in processes. In particular, the places `P1oDB` and `P2DB` represent the dirty bits of P_1^{old} and P_2, respectively. Each of those places may have a marking of zero or one, which can be interpreted as the confidence in the corresponding process. The timed and instantaneous activities together precisely represent how the MDCD checkpointing rule and AT-based validation policy are executed. For example, when being activated, the timed activity `P2_CKPT` indicates that P_2 is engaged in a checkpoint establishment, while the instantaneous activity `P2SkipCKPT` indicates that P_2 is exempted from checkpointing for a particular message receiving event, according to the MDCD checkpointing rule.

In order to evaluate the performance overhead reduction resulting from use of the dynamic confidence-driven approach, we conduct a numerical comparative study in the next subsection. Specifically, we contrast the performance overhead of the MDCD protocol with that of an MDCD variant employing a static confidence-driven approach. The static variant treats all the interacting processes in a system in which an application software component is undergoing an upgrade as low-confidence processes. In other words, all the processes, including P_1^{old} and P_2, are invariably regarded as potentially contaminated processes. Accordingly, in the same manner as P_1^{new}, P_2 applies AT to every external message it intends to send, regardless of whether P_2's process state is influenced by a not-yet-validated message from P_1^{new}. In addition, P_1^{old} and P_2 will perform checkpointing upon every message receiving event. Due to the pessimistic nature of this variant, we call it the *MDCD-P* protocol. By modifying the specifications of the input gates that define the conditions under which a process will perform checkpointing and AT, we adapt the SAN model shown in Fig. 8 for assessing performance overhead of this variant.

5.2 Evaluation Results

We focus our attention on the performance overhead resulting from applying the MDCD protocol during guarded operation. More precisely, we define the *performance overhead* as the expected fraction of time a process is performing checkpointing or AT. Using the SAN models developed in Section 5.1, this measure can be translated into the probability that at least one of the timed activities that represent that a particular process is performing checkpointing or AT-based validation is enabled in the steady state. Since the measure is defined on an individual process basis, we assign reward rates and compute the expected rewards separately as follows:

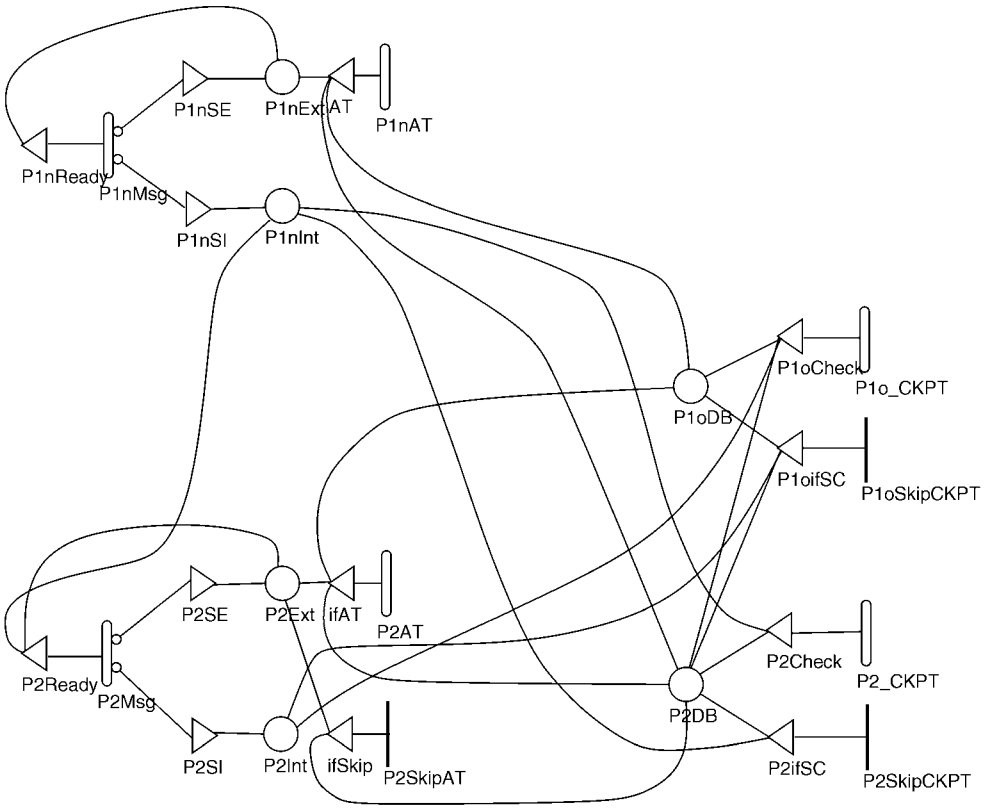


Fig. 8. SAN model for evaluating performance overhead.

- For P_1^{new}, we define a reward rate of 1 for each state of the SAN models in which the timed activity `P1nAT` is enabled, and compute the expected instant-of-time reward in steady state.
- For P_1^{old}, we define a reward rate of 1 for each state of the SAN models in which the timed activity `P1o_CKPT` is enabled and compute the expected instant-of-time reward in steady state.
- For P_2, we define a reward rate of 1 for each state of the SAN models in which the timed activity `P2AT` or `P2_CKPT` is enabled and compute the expected instant-of-time reward in steady state.

Before we proceed to discuss the evaluation results, we define the following notation:

σ_1^{new}: Performance overhead of P_1^{new} with the MDCD protocol.

σ_1^{old}: Performance overhead of P_1^{old} with the MDCD protocol.

σ_2: Performance overhead of P_2 with the MDCD protocol.

η_1^{new}: Performance overhead of P_1^{new} with the MDCD-P protocol.

η_1^{old}: Performance overhead of P_1^{old} with the MDCD-P protocol.

η_2: Performance overhead of P_2 with the MDCD-P protocol.

λ: Message sending rate of a process.

α: Acceptance-test completion rate.

β: Checkpoint-establishment completion rate.

p_{ext}: Probability that the message a process intends to send is an external message.

Since the major objective of the MDCD protocol is to mitigate the effect of software faults, checkpointing is normally conducted in the fast local memory instead of on disk. As explained in Section 3.2, ATs for embedded applications can usually be implemented in a more efficient way relative to those for generic application software. Accordingly, the parameter values used in the following analytic studies reflect those system characteristics.

As the first step, we study the performance overhead as a function of λ for the MDCD and MDCD-P protocols. The value assignment for other parameters is as follows (all parameters involving time presume that time is quantified in seconds):

$$\alpha = 100, \qquad \beta = 125, \qquad p_{ext} = 0.1$$

Fig. 9 thus compares, on an individual process basis, the performance overhead associated with the dynamic and static confidence-driven approaches. The numerical results of σ_1^{new} and η_1^{new} reveal that the performance overhead of P_1^{new} is insensitive to the type of confidence-driven approach (i.e., dynamic or static). This is a reasonable result because, throughout the guarded operation, P_1^{new} is invariably considered as potentially contaminated by the MDCD protocol, meaning that there is no confidence change in P_1^{new}; thus, the process does not gain performance benefit from the dynamic confidence-driven approach. However, since P_1^{new} is never involved in checkpointing activities, both σ_1^{new} and η_1^{new} are consistently lower than σ_2 and comparable

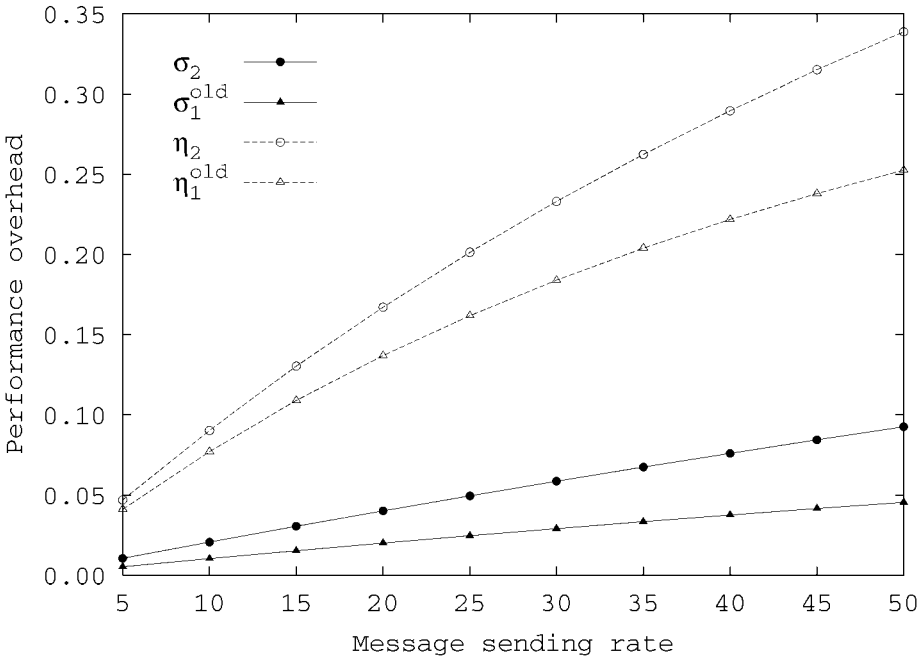


Fig. 9. Performance overhead as a function of λ.

to σ_1^{old}. For clarity of illustration, we do not display the numerical values of σ_1^{new} and η_1^{new} in Fig. 9 and the following figures.

An examination of Fig. 9 reveals that both P_2 and $\mathrm{P}_1^{\text{old}}$ obtain appreciable performance overhead reductions from use of the dynamic confidence-driven approach. In particular, performance overhead reductions become progressively more significant when message sending rate λ increases. The savings are primarily due to the checkpointing rule enforced by the MDCD protocol. In particular, with the dynamic confidence-driven approach, checkpoint establishment is required only when a message passing event degrades our confidence in a process. On the other hand, when the static confidence-driven approach is employed, a process must perform checkpointing upon every message receiving event; thus, its performance overhead goes up rapidly as the message sending rate λ increases. More specifically, after λ reaches 15 (equivalent to a scenario in which an individual process sends a message every 67 milliseconds on average), both η_2 and η_1^{old} exceed 0.10 and quickly become unacceptable. On the other hand, both σ_2 and σ_1^{old} are under 0.05 (well below η_2 and η_1^{old}) for the domain of λ up to 25 (equivalent to a scenario in which an individual process sends a message every 40 milliseconds on average), which is the typical message rate domain for the distributed embedded systems considered. As indicated by the curves, even when message passing events are bursty, i.e., the mean time between message sending events by an individual process becomes 20 milliseconds (i.e., $\lambda = 50$), σ_2 and σ_1^{old} remain below 0.10 and 0.05, respectively.

In Fig. 10, we illustrate the performance overhead as a function of the checkpoint-establishment completion rate β (the reciprocal of the mean time to completion of a checkpoint establishment). In this evaluation, the value of λ is set to 20; other parameter values remain the same as those used for the previous evaluation. The results shown in Fig. 10 further confirm that the MDCD approach effectively reduces performance overhead. In particular, when β is low (i.e., the performance cost for establishing a checkpoint is high), the overhead reduction is more significant—up to approximately 80 percent for both P_2 and $\mathrm{P}_1^{\text{old}}$.

In Fig. 11, performance overhead is plotted as a function of p_{ext}, the probability that a message that a process intends to send is an external message. Again, the numerical results show significant performance overhead reduction resulting from use of the dynamic confidence-driven approach. It is interesting to note that, while η_2 and η_1^{old} increase about linearly as p_{ext} increases, σ_2 and σ_1^{old} are clearly not linear functions of p_{ext}. In particular, σ_2 and σ_1^{old} increase slowly as p_{ext} increases in its lower domain; after p_{ext} reaches 0.25 and 0.40, further increase in p_{ext} results in decreases of σ_1^{old} and σ_2, respectively. This can be understood by considering the trade-off between the performance costs of error containment actions associated with internal and external message sending events. Generally, when the overall message sending rate of a process λ is fixed, a greater p_{ext} means that more AT-based validation activities may take place in the system. A greater p_{ext} also means that P_2 and $\mathrm{P}_1^{\text{old}}$ are more likely to receive a "passed AT" notification message between the receipts of application-purpose messages from a potentially contaminated process. This, in turn, causes the processes to perform checkpointing more frequently. On the other hand, an increase of p_{ext} also implies fewer internal message passing events so that potential process state contamination caused by interprocess communication becomes less likely to occur. Hence, the occurrence of confidence change for P_2 and $\mathrm{P}_1^{\text{old}}$ tends to come less often and checkpointing activities will become less frequent accordingly. This leads σ_2 and σ_1^{old} to increase slowly in the beginning and then decrease eventually.

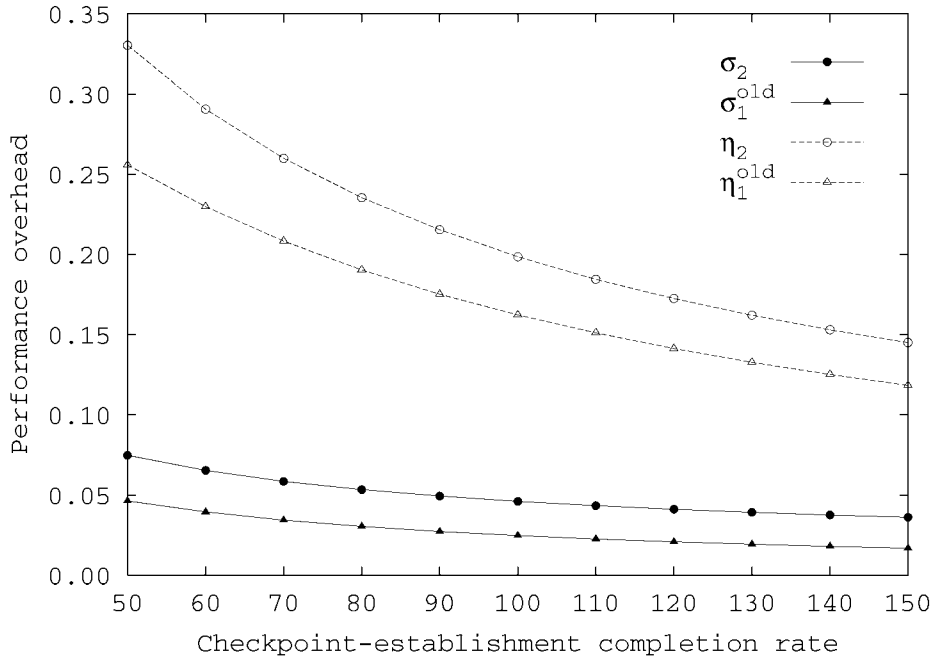


Fig. 10. Performance overhead as a function of β.

The above analysis demonstrates that the dynamic confidence-driven approach is superior to the pessimistic static confidence-driven approach in terms of performance overhead and the reliability evaluation we conducted earlier [9] reveals that the further reliability improvement resulting from use of the pessimistic MDCD variant is practically negligible. The analyses collectively not only confirm that the MDCD protocol is a low-cost, effective approach to onboard guarded software upgrading, but also reveal the potential of this approach to serve distributed embedded systems as a general-purpose, low-cost software fault tolerance technique.

6 Discussion

From a technical perspective, the core of the GSU framework, the MDCD protocol, is devised based on adaptation of several enabling techniques in the areas encompassing:

- checkpointing-based error recovery for message-passing distributed systems,

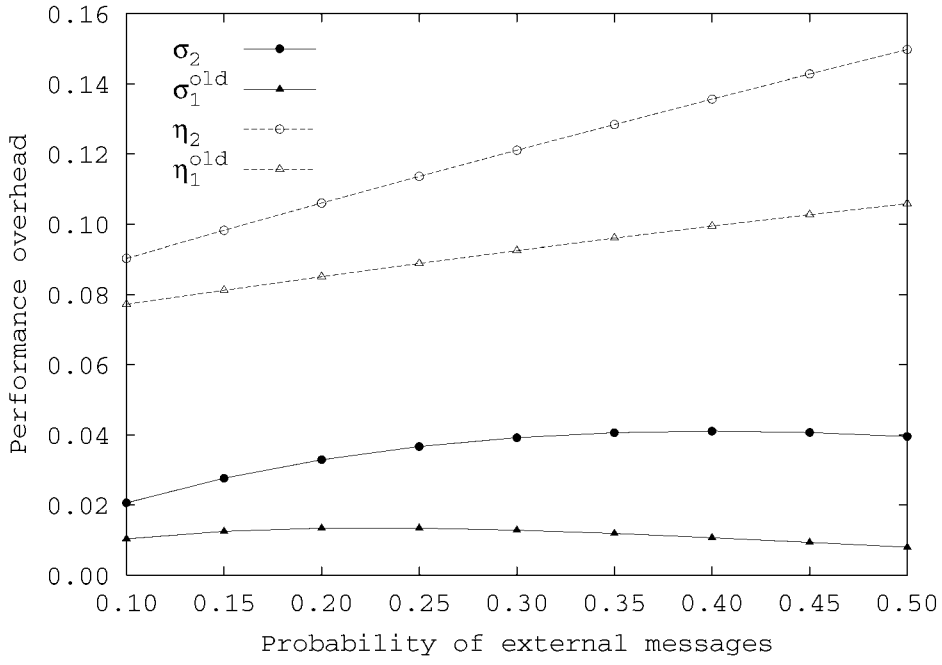


Fig. 11. Performance overhead as a function of p_{ext}.

- software fault tolerance in distributed computing environments, and
- confidence-oriented fault tolerance mechanisms.

In the following subsections, we compare the MDCD approach with the previous developments in the above areas. We also discuss the characteristics of the MDCD approach and its potential utility as a general-purpose, low-cost software fault tolerance technique for distributed embedded computing.

6.1 Checkpointing-Based Error Recovery

As discussed in the beginning of Section 4, in order to effectively mitigate the effects of software design faults in a distributed computing environment without imposing restriction on interprocess communication, we adapt the communication-induced checkpointing technique for hardware error recovery [7], [8] and complement the technique by introducing the confidence-driven notion. This is the most crucial step we take in deriving the distributed algorithms for low-cost error containment and recovery.

The resulting checkpointing rule and algorithms thus ensure that the error recovery mechanisms can bring the system into a global state that satisfies consistency and recoverability properties that concern verified correctness (validity) of process states and messages. Since our approach shares a significant portion of its theoretical basis with the checkpointing protocols for hardware fault tolerance, the MDCD protocol is able to coordinate with an existing time-based checkpointing protocol [18] in a synergistic fashion for simultaneous tolerance of software and hardware faults.

Although the derivation of the MDCD checkpointing rule is based on a particular type of distributed embedded system (i.e., a system that consists of two functionally different interacting software components, one of which has two functionally similar versions), the concepts are applicable to a general class of distributed systems. Accordingly, we have recently extended the checkpointing rule and algorithms such that the MDCD approach can be utilized as a general-purpose, low-cost software fault tolerance technique for distributed computing [20].

6.2 Software Fault Tolerance

In the area of software fault tolerance, much of the work, including NVP and DRB (distributed recovery blocks [13]), was intended to take advantage of redundant resources of a distributed system to support the concurrent execution of functionally equivalent, diverse software versions. The issues and problems concerning the effects of software design faults in a distributed computing environment received relatively less attention. Furthermore, traditional fault-tolerant software is usually prestructured, meaning that system behavior for checkpointing, result checking, output releasing, and error recovery is determined and structured during the design and development phases. This, in turn, prevents a software component from having dynamic interdependencies with other processes [13] and, thus, seriously limits the use of those software fault tolerance schemes in distributed computing environments.

Several research projects addressed the issues of software fault tolerance for distributed applications. The approaches suggested by those projects can be classified into two categories, namely, "coordination-by-programmer" (see [17], for example) and "coordination-by-machine" (see [21], for example). By prestructuring the application, "coordination-by-programmer" techniques let a pre-identified group of interacting processes by synchronized such that they will not proceed to communicate with other processes outside the group until all of the processes within the group pass acceptance tests or other types of error detection mechanisms. The second category, "coordination-by-machine" is also called "programmer-transparent-coordination." While relieving the application programmer of the burden of interacting processes, "coordination-by-machine" incurs additional performance costs (relative to "coordination-by-programmer") due to various types of coordination-purpose message exchange and data structure maintenance/search activities. The message-driven confidence-driven nature of our approach makes it different from traditional software fault tolerance in several respects. First, this approach does not impose any restrictions on interactions among application software components or require costly message-exchange-based process coordination/synchronization. Second, the MDCD algorithms allow us to provide fault tolerance to a critical software component only, while letting other components be protected against the effect of error propagation. Third, the dynamic error containment and recovery mechanisms are transparent to the application and thus can be implemented by generic middleware.

While the MDCD protocol is itself a means of software fault tolerance, the concept and approach can also be utilized to enable flexible, cost-effective use of traditional software fault tolerance schemes in distributed systems. In particular, schemes that are characterized by the concurrent execution of primary and secondary routines, such as NSCP (N-self-checking programming [22]), can be readily accommodated by the MDCD approach. More specifically, with the MDCD protocol, the NSCP scheme can be applied to a low-confidence component only while permitting it to interact with other components with no restrictions.

6.3 Confidence-Oriented Fault Tolerance Mechanisms

To the best of our knowledge, there has been no prior effort to investigate the combined message-driven confidence-driven approach. A few software fault tolerance projects investigated confidence-oriented fault tolerance mechanisms. For example, in an NVP project, it was proposed to let the decision algorithm employ a "gold version" (i.e., a version that deserves higher confidence relative to other versions of the same program) as a reference for detecting erroneous voting results caused by related faults [23]. In [13], it was suggested that the secondary (backup) routine in the RB or DRB scheme could use a version that is not as efficient as, but is more reliable than, the primary (active) routine. In contrast with those proposed mechanisms, the MDCD approach enables us to adjust our confidence in a process dynamically at runtime by keeping track of possible error propagation due to message passing.

More recently, an integrity policy was proposed to mediate the information flows between objects with

different levels of integrity (where "integrity" is defined as the trustworthiness with respect to an object's correctness and the consequence of its failure) [24], [25]. Normally, the integrity policy is intended to prohibit information flows from a low integrity object to a high integrity object. But, when such information flow is necessary for an application, validation and fault tolerance mechanisms will be invoked to make sure that no erroneous information from the object of low integrity will corrupt the object of high integrity.

Although both approaches allow software components with different confidence/integrity levels to coexist in a dependable system, our confidence-driven approach differs from the integrity policy based mechanism in a number of respects. First, the integrity policy-based mechanism aims at avoiding error propagation at the program control/information flow level, while the MDCD protocol deals with potential error contamination at the system level at which processes execute in an asynchronous fashion and cooperate through message passing. Thus, with the MDCD approach, confidence in a particular process may naturally be degraded and upgraded, through message passing and validation events, respectively, during guarded operation. A more significant difference between the two approaches is that while the integrity policy is intended to *prevent*, by controlling and mediating the information flow, erroneous information from affecting an object, our approach *allows* the interacting processes to talk to each other without restriction but keeps track of potential error contamination to enable recovery actions. Accordingly, we provide AT-based validation only at the system boundary (across which an error cannot be recovered through rollback) and make use of the test result to judge the validity of the states of individual processes in the system. The motivation is to reduce performance cost by minimizing intervention in application-purpose interprocess communication.

7 Concluding Remarks

We have devised a low-cost approach to mitigating the effect of software design faults in a distributed embedded computing environment. Unlike the existing software fault tolerance schemes for distributed systems, the MDCD protocol is dynamic and asynchronous in nature, requiring no atomic action and permitting interacting processes to communicate with each other without restriction.

The dynamic message-driven confidence-driven approach not only plays a key role in performance overhead reduction, but also eliminates the need to prestructure application programs and thus facilitates an application-independent middleware implementation. Recently, we have completed the first version of the GSU Middleware which implements a prototype MDCD protocol [26]. This version includes 1) a group of MDCD modules responsible for assisting individual processes to maintain knowledge about potential process state contamination and to make decisions on whether to take a checkpoint upon message passing and whether to roll back or roll forward during recovery and 2) a set of invocable services (implemented as distributed objects in C++) that execute message sending or receiving requests from application processes in a manner that adapts to the confidence in the sending and/or receiving processes. The "overloading" characteristics of the invocable services allow the MDCD protocol to be transparent to the application programmer and the middleware itself to be generic and modifiable. Note that the only application-dependent element in the MDCD protocol is the ATs. And, in addition, the requirement that the message passing behavior of the redundant versions (e.g., P_1^{new} and P_1^{old}, for the GSU application) be equivalent is a common limitation of diverse-version based software fault tolerance techniques.

A number of factors other than upgrading may lead us to discriminate among interacting software components in a distributed system with respect to our confidence in their trustworthiness. For example, we may have low confidence in a software component with high complexity or poor testability. Software components in a distributed system may thus be categorized into two groups according to our confidence in their trustworthiness. Moreover, the constraints on power, mass, and performance overhead imposed on the avionics system we address in this paper are typical concerns of distributed embedded systems. The above factors, combined with the MDCD protocol's ability to facilitate the application of software fault tolerance to selected interacting components, suggest that the MDCD approach can be utilized as a general-purpose low-cost software fault tolerance technique for distributed embedded computing. As mentioned in Section 6, we have recently extended the MDCD algorithms by removing the architectural restrictions on the underlying system.

When a general class of distributed embedded systems is considered, the maintenance of knowledge about potential process state contamination becomes more challenging. In particular, when multiple high-confidence software components and/or multiple low-confidence components are present in a distributed system, individual processes may be potentially contaminated by different messages from a particular low-confidence component. Moreover, different processes in a system can be contaminated by errors in different low-confidence components. These factors collectively make the adjustment of confidence in individual processes more complex. To deal with the increased complexity, we introduce a *fine-grained* approach for adjusting confidence in a process state. By carefully adapting the checkpointing rule of the original MDCD protocol, we are able to permit a process to be validated partially and progressively and keep the performance overhead for the fine-grained confidence adjustment low. Specifically, with the extended algorithms, the view of the most recent noncontaminated state of a potentially contaminated process P is kept updated, based on fine-grained confidence adjustment, during the interval after the state of P becomes potentially contaminated and before P is validated as having a noncontaminated state or confirmed as being actually erroneous. This permits the interacting processes to roll back minimum distances to reach a consistent global state when error recovery is invoked. The carefully modified checkpointing rule not only enables fine-grained confidence adjustment, but also 1) prevents a process from establishing checkpoints that are "predictably useless" and 2) ensures timely removal

of checkpoints that become useless after fine-grained confidence adjustment. (The algorithm extension leads the checkpointing rule presented in Section 3 of this paper to become a boundary-case version of the generalized checkpointing rule.)

Our current work is directed toward formalizing the generalized message-driven confidence-driven approach. In order to experimentally validate the feasibility of applying the MDCD approach to enable low-cost flexible software fault tolerance in distributed embedded systems, we have been implementing fault injection mechanisms. We are also in the process of porting the prototype GSU Middleware to the Future Deliveries Testbed at JPL.

ACKNOWLEDGMENTS

The work reported in this paper was supported in part by NASA Small Business Innovation Research (SBIR) Contract NAS3-99125. Portions of the material presented in this paper were presented in an earlier form at the 20th International Conference on Distributed Computing Systems (ICDCS 2000).

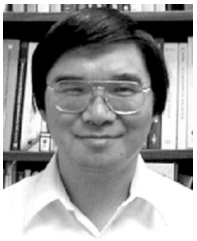
REFERENCES

[1] R. Baalke, "Mars Pathfinder Update," Mars Pathfinder Weekly Status Report, Office of the Flight Operations Manager, Jet Propulsion Laboratory, California Inst. of Technology, Pasadena, June 1997.
[2] J. Rendleman, "MCI WorldCom Blames Lucent Software for Outage," *PC Week,* 16 Aug. 1999.
[3] L. Sha, J.B. Goodenough, and B. Pollak, "Simplex Architecture: Meeting the Challenges of Using COTS in High-Reliability Systems," *CrossTalk: The J. Defense Software Eng.,* vol. 11, pp. 7-10, Apr. 1998.
[4] D. Powell et al., "GUARDS: A Generic Upgradable Architecture for Real-Time Dependable Systems," *IEEE Trans. Parallel and Distributed Systems,* vol. 10, no. 6, pp. 580-599, June 1999.
[5] M.E. Segal and O. Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software,* vol. 10, no. 2, pp. 53-65, Mar. 1993.
[6] A.T. Tai and K.S. Tso, "On-Board Maintenance for Affordable, Evolvable and Dependable Spaceborne Systems," Phase-I Final Technical Report for Contract NAS8-98179, IA Tech, Inc., Los Angeles, Oct. 1998.
[7] E.N. Elnozahy, D.B. Johnson, and Y.-M. Wang, "A Survey of Rollback-Recovery Protocols in Message-Passing Systems," Technical Report CMU-CS-96-181, School of Computer Science, Carnegie Mellon Univ., Pittsburgh, Penn., Oct. 1996.
[8] Y.M. Wang et al., "Checkpointing and Its Applications," *Digest 25th Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 22-31, June 1995.
[9] A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau, and W.H. Sanders, "On the Effectiveness of a Message-Driven Confidence-Driven Protocol for Guarded Software Upgrading," *Performance Evaluation,* vol. 44, pp. 211-236, Apr. 2001.
[10] S.N. Chau, L. Alkalai, A.T. Tai, and J.B. Burt, "Design of a Fault-Tolerant COTS-Based Bus Architecture," *IEEE Trans. Reliability,* vol. 48, pp. 351-359, Dec. 1999.
[11] C.T. Baker, "Effects of Field Service on Software Reliability," *IEEE Trans. Software Eng.,* vol. 14, no. 2, pp. 254-258, Feb. 1988.
[12] A.T. Tai, K.S. Tso, L. Alkalai, S.N. Chau, and W.H. Sanders, "On Low-Cost Error Containment and Recovery Methods for Guarded Software Upgrading," *Proc. 20th Int'l Conf. Distributed Computing Systems (ICDCS 2000),* pp. 548-555, Apr. 2000.
[13] K.H. Kim, "The Distributed Recovery Block Scheme," *Software Fault Tolerance,* M.R. Lyu, ed., pp. 189-209, West Sussex, England: John Wiley & Sons, 1995.
[14] H. Wasserman and M. Blum, "Software Reliability via Run-Time Result-Checking," *J. ACM,* vol. 44, no. 6, pp. 826-849, 1997.
[15] S. Edwards, L. Lavagno, E.A. Lee, and A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation, and Synthesis," *Proc. IEEE,* vol. 85, pp. 366-390, Mar. 1997.
[16] A. Avizienis, "The N-Version Approach to Fault-Tolerant Software," *IEEE Trans. Software Eng.,* vol. 11, no. 12, pp. 1491-1501, Dec. 1985.
[17] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Trans. Software Eng.,* vol. 1, pp. 220-232, June 1975.
[18] N. Neves and W.K. Fuchs, "Coordinated Checkpointing without Direct Coordination," *Proc. Third IEEE Int'l Computer Performance and Dependability Symp.,* pp. 23-31, Sept. 1998.
[19] W.H. Sanders, W.D. Obal II, M.A. Qureshi, and F.K. Widjanarko, "The *UltraSAN* Modeling Environment," *Performance Evaluation,* vol. 24, no. 1, pp. 89-115, 1995.
[20] A.T. Tai and K.S. Tso, "MDCD Algorithm Extension for a General Class of Distributed Embedded Systems," Phase-II Fifth Interim Technical Progress Report for Contract NAS3-99125, IA Tech, Inc., Los Angeles, June 2001.
[21] K.H. Kim, "Programmer Transparent Coordination of Recovering Concurrent Processes: Philosophy and Rules of Efficient Implementation," *IEEE Trans. Software Eng.,* vol. 14, no. 6, pp. 810-821, June 1988.
[22] J.-C. Laprie, J. Arlat, C. Béounes, and K. Kanoun, "Definition and Analysis of Hardware-and-Software Fault-Tolerant Architectures," *Computer,* vol. 23, no. 7, pp. 39-51, July 1990.
[23] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *Computer,* vol. 17, pp. 67-80, Aug. 1984.
[24] T. Vardanega, P.D.J.-F. Chane, W.M.R. Messaros, and J. Arlat, "On the Development of Fault-Tolerant On-Board Control Software and Its Evaluation by Fault Injection," *Digest 25th Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 510-515, June 1995.
[25] E. Totel, J.-P. Blanquart, Y. Deswarte, and D. Powell, "Supporting Multiple Levels of Criticality," *Digest 28th Ann. Int'l Symp. Fault-Tolerant Computing,* pp. 70-79, June 1998.
[26] K.S. Tso, A.T. Tai, L. Alkalai, S.N. Chau, and W.H. Sanders, "GSU Middleware Architecture Design," *Proc. Fifth IEEE Int'l Symp. High Assurance Systems Eng.,* pp. 212-215, Nov. 2000.

Ann T. Tai received the PhD degree in computer science from the University of California, Los Angeles. She is the president of and a senior scientist at IA Tech, Inc., Los Angeles. Prior to 1997, she was associated with SoHaR Incorporated as a senior research engineer. She was an assistant professor at the University of Texas at Dallas during 1993. Her current research interests concern the design, development, and evaluation of dependable computer systems, error containment and recovery algorithms for distributed computing, and distributed fault-tolerant system architectures. She authored the book, *Software Performability: From Concepts to Applications* (Kluwer Academic). She is a member of the IEEE.

Kam S. Tso received the PhD degree in computer science from the University of California, Los Angeles, the MS degree in electronic engineering from the Philips International Institute, Eindhoven, The Netherlands, and the BS in electronics from the Chinese University of Hong Kong, Hong Kong. From 1986 to 1996, he worked at the Jet Propulsion Laboratory and SoHaR Incorporated, conducting research and development on robotics systems, fault-tolerant systems, and reliable software. He is currently vice president of IA Tech, Inc. Dr. Tso's research interests include World Wide Web technologies, distributed planning and collaboration, high performance, and dependable real-time software and systems. He is a member of the IEEE.

Leon Alkalai received the PhD degree in computer science in 1989 from the University of California, Los Angeles after which he joined the Jet Propulsion Laboratory (JPL), California Institute of Technology, where he is the center director for the Center for Integrated Space Microsystems, a Center of Excellence at JPL. The main focus of the center is the development of advanced microelectronics, micro-avionics, and advanced computing technologies for future deep-space highly miniaturized, autonomous, and intelligent robotic missions. He has worked on numerous technology development tasks, including advanced microelectronics miniaturization, advanced microelectronics packaging, and reliable and fault-tolerant architectures. He was also one of the NASA appointed coleads on the New Millennium Program Integrated Product Development Teams for Microelectronics Systems, a consortium of government, industry, and academia to validate technologies for future NASA missions in the 21st century. He is a member of the IEEE.



Savio N. Chau received the PhD degree in computer science from the University of California, Los Angeles. He is principle engineer and the supervisor of the Advanced Concepts and Architecture Group at the Jet Propulsion Laboratory. He is currently developing scalable multimission avionics system architectures. He has been investigating techniques to apply low-cost commercial bus standards and off-the-shelf products in highly reliable systems such as long-life spacecraft. His research areas include scalable distributed system architecture, fault tolerance, and design-for-testability. He is a member of Tau Beta Pi and Eta Kappa Nu. He is a member of the IEEE.

William H. Sanders received the BSE degree in computer engineering (1983), the MSE degree in computer, information, and control engineering (1985), and the PhD degree in computer science and engineering (1988) from the University of Michigan. He is currently a professor in the Department of Electrical and Computer Engineering and the Coordinated Science Laboratory at the University of Illinois at Urbana-Champaign. He is chair of the IEEE TC on Fault-Tolerant Computing and vice-chair of IFIP Working Group 10.4 on Dependable Computing. In addition, he serves on the Board of Directors of ACM Sigmetrics and the editorial board of the *IEEE Transactions on Reliability*. He is a fellow of the IEEE and a member of the IEEE Computer, Communications, and Reliability Societies, as well as the ACM, IFIP Working Group 10.4 on Dependable Computing, Sigma Xi, and Eta Kappa Nu. Dr. Sanders's research interests include performance/dependability evaluation, dependable computing, and reliable distributed systems. He has published more than 100 technical papers in these areas. He was program cochair of the 29th International Symposium on Fault-Tolerant Computing (FTCS-29), was program cochair of the Sixth IFIP Working Conference on Dependable Computing for Critical Applications, and has served on the program committees of numerous conferences and workshops. He is a co-developer of three tools for assessing the performability of systems represented as stochastic activity networks: METASAN, *UltraSAN*, and Möbius. *UltraSAN* has been distributed widely to industry and academia and licensed to more than 200 universities, several companies, and NASA for evaluating the performance, dependability, and performability of complex distributed systems. He is also a co-developer of the Loki distributed system fault injector and the AQuA middleware for providing dependability to distributed object-oriented applications.